Fast Arbitrary Precision Integer Multiplication

by Henrik Vestermark (hve@hvks.com)

Abstract

This paper surveys the primary methods for multiplying very large integers (arbitrary precision numbers), from the simple grade-school algorithm, linear convolution, to Karatsuba, Toom–Cook, Fast Fourier Transform (FFT-based), and Number-Theoretic Transform (NTT) approaches, on to Schönhage–Strassen and Fürer's method. We show how to store big numbers in C++ using the std::vector<uintmax_t> library and outline practical implementation details such as carry handling and chunk sizing. We provide pseudocode for each algorithm, explain when it outperforms simpler methods, and offer C++ code snippets. We also present benchmark results on modern 64-bit hardware to guide the choice of algorithm by operand size.

Introduction to Why Fast Multiplication Matters

Multiplying two 64-bit numbers is a single CPU instruction. Still, many modern tasks, such as RSA keys, elliptic-curve arithmetic, homomorphic encryption, and long-running number-theory projects, work with numbers that are thousands, millions, or even billions of bits long. At that size, the straightforward "school" method becomes far too slow because the work grows with the square of the size of the numbers.

With millions of digits to multiply, the implementation of a multiplication algorithm needs to be as fast as possible. The way we learn multiplication in elementary school can be applied to larger operands, and it will still produce the correct result for very large decimal numbers. However, the drawback is speed. The school algorithm is an $O(n^2)$ method. Meaning that if you double the number of digits, the workload increases by the square amount. Until the beginning of the 1960s, this was believed to be the limitation. However, Karatsuba proved in 1962 that multiplication could be performed with a complexity of $O(n^{1.58})$ [1], which was later improved using the Toom-Cook method [2] to achieve a complexity of as low as $O(n^{1.48})$. Later on, it was discovered that using FFT (Fast Fourier Transformation), the complexity could be reduced to $O(n \cdot \log(n))$. And again in 1972, Schönhage-Strassen, the O() complexity was reduced further.
This paper will discuss the practical implementation of some of these algorithms, beginning with basic elementary school multiplication and then progressing to more advanced methods.

In most modern computer languages, there is usually not a single type like the C++ int type that can hold more than a maximum of 64-bit integers. That is insufficient to hold more than around 19 decimal digits. If you need to go higher, you must have a way to store these huge numbers in a structure. Fortunately, the C++ standard library provides a template vector structure that can hold arbitrary elements. In the code base for this project, every integer is kept in:

```
std::vector<uintmax_t> limbs;   // limbs[0] holds the least-significant 64 bits
```

In most C++ implementations, the type uintmax_t is a 64-bit unsigned integer of 64bits.

All the algorithms described below act on that layout. In practice, you will combine several algorithms in an implementation depending on the number of digits of the operand of the multiplication: a simple one for tiny inputs, a divide-and-conquer one for mid-sized inputs, and some flavor of FFT or NTT for the truly huge cases.

Contents

Abstract 1
Introduction to Why Fast Multiplication Matters 1
Using std::vector<uintmax_t> for Arbitrary-Precision Integers 4
  Why a vector of 64-bit words? 4
  Why dynamic allocation? 4
  The role of reserve() in avoiding slow growth 5
  Appending elements: push_back() vs emplace_back() 5
  Accessing limbs efficiently 5
  Use std::vector<uintmax_t> 5
Multiplication 6
  Elementary school multiplication 6
  Linear convolution multiplication 8
    No Splitting, No FFT – Just Multiply and Add 9
    The Algorithm 9
    The 64-bit Case 10
    Carry Propagation: Why It's Done This Way 11
    Comparison to 32-bit Versions 11
    Benefits of Linear Convolution 11
    Performance of Linear Convolutions 12
  Karatsuba multiplication 12
    Performance of Karatsuba 15
  Toom-Cook multiplication 16
    Performance of ToomCook3 19
  Fast Fourier Transformation (FFT) 20
    Step 1. Unpacking a and b into fa and fb. 24
    Step 2. What happens in fa = FFT(a) and fb = FFT(b)? 24
    Why do we need the bit-reversal before the main Cooley–Tukey loops? 25
    Step 3. Why can we multiply fa[i] * fb[i] pointwise? 26
    Step 4. What does the inverse FFT(fc) do? And why carry propagation? 26
    Carry propagation 27
    Why does FFT radix-2 require a power-of-two input length? 27
    Why do we require input size ≥ 2^n, and then double it again (n <<= 1)? 28
    Can you stop carrying propagation after n + m - 1 digits? 29
    Limitations of the FFT 29
    Performance of FTT 31
  NTT method 31
    NTT Code Comments 38
    Step 1. Pick your prime and padding length 38

Step 2. Forward NTT (the "frequency domain") 38
Step 3. Pointwise multiply 38
Step 4. Inverse NTT (back to "time domain") 38
Step 5. Repeat for all three primes 38
Step 6. Reconstruct the true coefficients via CRT 39
The Chinese Remainder Theorem (CRT) 39
Step 1. Prepare your primes and inverses for CRT 39
Step 2. First merge: combine r_0 and r_1 39
Step 3. Final merge: combine u_i with r_2 39
Why the CRT works 40
Pros and Cons of NTT vs FFT Method 40
NTT Limitations with Primes less than 2^{32} 41
Why do we need multiple primes to make NTT useful? 41
NTT Limitations with Primes less than 2^{64} 41
Performance: Why three 32-bit primes can outperform a single 64-bit prime 41
NTT Performance of 32-bit and 64-bit 42
Schönhage-Strassen Theory vs Practice 43
The Mathematical Foundation 43
Complexity Analysis 43
The Recursive Structure 44
The Implementation Challenge 44
Ring Size Explosion 44
Precision and Numerical Stability 45
Practical Compromises and Hybrid Approaches 45
The Bootstrap Problem in Practice 46
Cache Behavior and Memory Hierarchy 46
Modern Library Implementations 46
The Theory-Practice Divide 47
Fürer's Method 47
Choosing the right method in real projects 47
Performance 48
Conclusion 49
Recommendation for arbitrary multiplication implementation 49
Reference 49

Using std::vector<uintmax_t> for Arbitrary-Precision Integers

Working with very large integers, which can be millions or billions of bits, requires a data structure that is both flexible and efficient. In this project, we represent an arbitrary-precision integer using a C++ std::vector<uintmax_t>, where each element (or “limb”) holds a fixed-width chunk of the full number, and limbs[0] is always the least significant limb.

This chapter explains why this choice is practical and efficient, and how to use it correctly to avoid unnecessary slowdowns.

Why a vector of 64-bit words?

Using a dynamic array of unsigned integers is one of the most common strategies for representing large integers. The std::vector<> container is a natural fit for this task:

```
std::vector<uintmax_t> limbs;
```

On modern systems, uintmax_t usually resolves to uint64_t, giving you 64-bit limbs. This size is ideal for several reasons:

- It's the native word size on 64-bit CPUs, so operations like addition, multiplication, and bit shifts are efficient.
- You can take advantage of hardware instructions like mul, adc, sbb, and shld that operate on 64-bit words.
- It balances granularity and capacity, 32-bit limbs waste memory and cause more carry steps, while 128-bit limbs are not widely supported without compiler extensions.

You could argue for std::vector<uint64_t> directly. That’s also fine, and in this codebase, they are essentially interchangeable, as we usually alias uintmax_t for clarity and potential portability.

Why dynamic allocation?

Arbitrary-precision multiplication often grows the number of limbs beyond the input size. For example, multiplying two n-limb numbers can produce up to $2n$ limbs. A dynamically sized container like std::vector handles this growth automatically.

This lets you:

- Start with just a few limbs for small inputs.
- Grow naturally when the number increases.
- Avoid tracking buffer sizes manually.

Static arrays would either waste space or fail unpredictably when the number of elements exceeded the fixed size.

The role of reserve() in avoiding slow growth

By default, std::vector grows its internal buffer when needed, but this involves reallocating and copying, which is expensive in a performance-critical algorithm like large integer multiplication.

Use the vector method reserve() to allocate enough space up front:

```
std::vector<uintmax_t> result;
result.reserve(a.size() + b.size());  // avoid reallocation during multiply
```

This avoids repeated reallocation and copying when calling push_back() or writing to elements by index.

Pre-reserving space can significantly improve performance in recursive algorithms, especially Karatsuba or Toom-Cook, where many intermediate vectors are created.

Appending elements: push_back() vs emplace_back()

When adding new limbs to the vector, there are two common methods:

```
v.push_back(x);
v.emplace_back(x);
```

For fundamental types like uintmax_t, these do the same thing. emplace_back() is useful when the vector stores more complex objects (like structs or classes) since it constructs the object in place.

But for simple integers:

- push_back() is clear and slightly more idiomatic.
- emplace_back() has no advantage in this context.

Therefore, there is no practical difference here. Choose whichever you prefer, but push_back() is usually more readable when adding plain numbers.

Accessing limbs efficiently

You can access limbs directly with operator[] or use at() if you want bounds checking:

```
uintmax_t low = limbs[0];               // fast, unchecked
uintmax_t check = limbs.at(1);          // slow, throws if out-of-bounds
```

For performance-critical code, [] is preferred. Just be sure the index is valid. Most algorithms maintain strict control over vector sizes, so bounds checks are unnecessary in inner loops.

Also, remember that vector size does *not* always equal the actual number's digit length, trailing zero limbs may be trimmed for normalization.

Use std::vector<uintmax_t>

Using std::vector<uintmax_t> is an effective way to manage large integers in C++. It provides:

- Dynamic growth for numbers of any size.

- Clean syntax and STL integration.
- Efficient storage using 64-bit limbs.

To get the most out of it:

- Call reserve() before filling the vector to avoid reallocations.
- Stick with push_back() for appending values.
- Use [] rather than at() inside performance-sensitive code.

In short, std::vector gives you a flexible and efficient base that works well with modern C++ practices. It allows your multiplication algorithms to scale up as far as your hardware will allow.

Multiplication

Multiplication is trivial. Now that we have settled on a vector<uintmax_t> to represent the number. We need to examine how multiplication works with two uintmax_t elements. e.g., uintmax_t a,b; when multiplying a·b, the correct result is a 128-bit unsigned integer. However, in C++, the highest 64 bits of the result are discarded since there is no larger integer representation in C++. We would therefore need to implement a check of a 64-bit overflow result and recover the highest 64 bits of the result to be used as a carry for the next limb. For many of the source codes listed, you may have to consult the actual source code since there will be subfunctions called that are not listed in the code segment in this paper. In the code segment, the type iptype has been aliased with the uintmax_t type.

Elementary school multiplication

For the multiplication, we divide the case into two scenarios: one where the operand b contains a single limb, m=1 (64-bit integer), and one where m>1. For m=1, we use the forward loop:

Algorithm: Unsigned multiplication with a single limb digit

```
Multiply (vector a, limb b)
BASE=2^64       // Our Base number is a single limb
carry=0         // Zero the carry
for(i=0..n)     // n is the number of limbs in a vector, c is the result vector
        c[i]=(a[i]*b+carry)%BASE
        carry=(a[i]*b+carry)/BASE
if(carry!=0)
        c[n+1]=carry
```

For m>1, we repeatedly use the above formula for multiplying a single digit and adding the intermediate results.

Algorithm: unsigned Multiplication of two vectors a and b.

```
Multiply (vector a, vector b)
BASE=2^64
c_k=a_n*b[0]              // Single digit multiplication
for(i=1..m)
        tmp=a_n*b[i]     // Single digit multiplication
        c_k=c_k+tmp*BASE^i      // BASE^i is the offset
```

Note. The carry handling is omitted in the above algorithm.

Multiplying the intermediate result by $BASE^i$ is easy, as you postfix the temporary result with *i* number of zeros. The above algorithm has a complexity of $O(n^2)$ and is typically referred to as elementary school multiplication.

```
std::vector<iptype> _int_precision_umul_short(const std::vector<iptype>& src1, const iptype
d, const size_t offset)
        {
        iptype carry=0;
        std::vector<iptype>::const_iterator pos, end;
        std::vector<iptype> des;
        std::vector<iptype> tmp(2);

        des.reserve(src1.capacity()+offset);  // Reserve space to avoid time consuming
reallocation
        // Preallocate space for zero offset entries into des.
        if(offset!=0) des.insert(des.end(), offset, 0);

        if (d == 0)  // Multiply by zero is zero.
                {
                des.push_back(0);
                return des;
                }

        if (d == 1)  // Multiply by one dont change the src1.
                {
                des.insert(des.end(), src1.begin(), src1.end());
                //_int_precision_strip_trailing_zeros(des);
                return des;
                }

        pos = src1.begin();
        end = src1.end();
        for (; pos != end; ++pos)
                {
                tmp=_precision_umul64(*pos, d);
                tmp[0] += carry;
                carry = (tmp[0] < carry) ? tmp[1]+1 : tmp[1]; // Set carry if overflow is
detected
                des.push_back(tmp[0]);
                }

        if (carry != 0)
                des.push_back(carry);
        _int_precision_strip_trailing_zeros(des);

        return des;
        }

std::vector<iptype> _int_precision_umul_school(const std::vector<iptype>& src1, const
std::vector<iptype>& src2)
        {
        int disp;
        std::vector<iptype> des, tmp;
        std::vector<iptype>::const_iterator pos2, pos2_end;
```

```
pos2 = src2.begin();
pos2_end = src2.end();
des = _int_precision_umul_short(src1, *pos2);
for (pos2++, disp = 1; pos2 != pos2_end; disp++, pos2++)
        {
        if (*pos2 != 0)
                {
                tmp = _int_precision_umul_short(src1, *pos2,disp);
                des = _int_precision_uadd(des, tmp);
                }
        }

_int_precision_strip_trailing_zeros(des);
return des;
}
```

Schoolbook multiplication is not the fastest way to do multiplication and is easily beaten by the performance of other multiplication methods. A few others are relevant to consider for multiplication:

- Linear convolution
- Karatsuba
- Toom-Cook 3
- Fast Fourier Transformation (FFT)
- Number-Theoretic Transform (NTT)
- Schönhage-Strassen
- Fürer's method

Linear convolution multiplication

Linear convolution is one of the most straightforward and exact ways to multiply two arbitrary-precision integers. The idea is simple: treat the two input vectors as polynomial coefficients and compute the coefficient-wise product directly, accounting for carry propagation.

Each input vector stores the number in base 2^{64}, using a standard format where std::vector<uintmax_t> v[0] is the least significant 64-bit word (limb), and the value of the full number is:

$$A = \sum_{j=0}^{n-1} a_j \cdot 2^{64}$$

Multiplying two such vectors correspond to computing the full polynomial product:

$$C_k = \sum_{i+j=k} a_i \cdot b_j$$

This is precisely what linear convolution computes, and the result has a maximum length of n + m limbs, assuming inputs of size n and m digits.

No Splitting, No FFT – Just Multiply and Add

Unlike divide-and-conquer methods (Karatsuba, Toom-Cook) or fast transforms (FFT/NTT), this approach doesn't require splitting the input vectors or transforming them into another domain. It processes the multiplication directly, using nested loops and explicit carry tracking.

This has two advantages:

- Simplicity. It's easy to reason about, debug, and get correct.
- Accuracy. There's no rounding error, no need for fixed-point tuning, and no modular arithmetic.

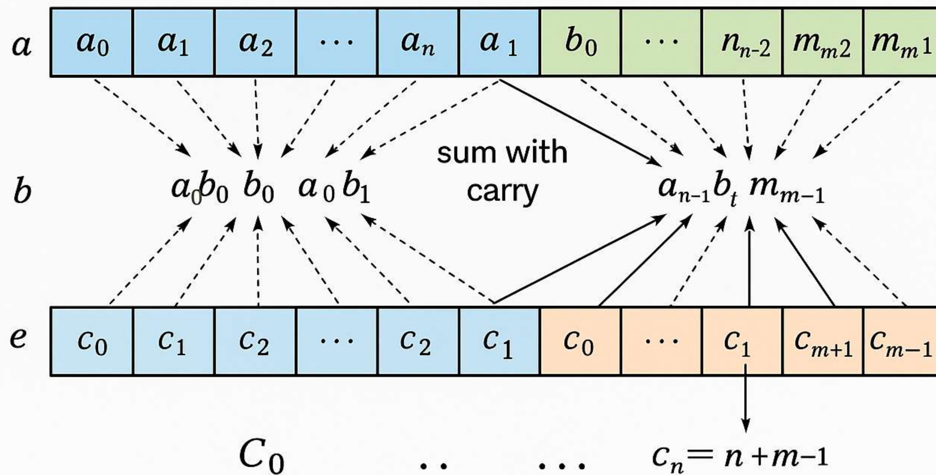
The tradeoff is performance: the algorithm runs in $O(n \cdot m)$ time, which becomes expensive as the inputs grow. However, it's still a good choice for small to medium-sized multiplications or as the base case of recursive methods.

The Algorithm

Below is a high-level description of the algorithm in pseudocode. For each combination of $a[j] \cdot b[i]$, we compute a 128-bit product and carefully add it to the result vector using portable 64-bit additions with explicit carry handling.

Algorithm: Linear convolution

```
function linear_convolution(vector vec_a, vector vec_b)
    size_a=size(vec_a)
    size_b=size(vec_b)
    for(i=0..size_b)
        for(j=0..size_a)
            tmp = umul64(vec_a[j], vec_b[i]); // Return 128bit result
            // portable 64-bit add with explicit carries
            s1 = tmp.lo + res[i + j];
            c1 = (s1 < tmp.lo);        // carry from first add
            s2 = s1 + carry;
            c2 = (s2 < carry);         // carry from second add
            res[i + j] = s2;
            carry = tmp.hi + c1 + c2;
            res[i + n] += carry;
        // propagate carry to res[i + size_a] and beyond if needed
    return res
```



Above is a graphic depicting what is going on.

The 64-bit Case

On a 64-bit architecture, 64-bit limbs (uint64_t) are ideal because the hardware directly supports multiplication and addition instructions. Here's the C++ implementation used in this project. (iptype is a 64bit unsinged integer).

```
std::vector<iptype>  _int_precision_umul_linear(const std::vector<iptype>&
lhs,const std::vector<iptype>& rhs)
{
        if (lhs.empty() || rhs.empty()) return { 0 };

        const std::size_t n = lhs.size();
        const std::size_t m = rhs.size();
        std::vector<iptype> res(n + m, 0);        // fully pre-sized

        for (std::size_t i = 0; i < m; ++i)
        {
                iptype carry = 0;

                for (std::size_t j = 0; j < n; ++j)
                {
                        mul128 tmp = _precision_umul64(lhs[j], rhs[i]);

                        // portable 64-bit add with explicit carries
                        iptype s1 = tmp.lo + res[i + j];
                        iptype c1 = (s1 < tmp.lo);          // carry from first add
                        iptype s2 = s1 + carry;
                        iptype c2 = (s2 < carry);       // carry from second add
                        res[i + j] = s2;
                        carry = tmp.hi + c1 + c2;
```

```
            }
            //res[i + n] += carry;                  // write final carry limb
            // propagate the leftover carry as far as needed
            std::size_t k = i + n;
            while (carry) {
                  iptype old = res[k];
                  res[k] += carry;
                  carry = (res[k] < old);   // 1 → overflow, keep rippling
                  ++k;
            }
      }

      _int_precision_strip_trailing_zeros(res);
      return res;
}
```

Carry Propagation: Why It's Done This Way

Rather than just writing res[i + n] += carry after the inner loop, the code uses a while loop to propagate the carry as far as necessary. This is important because: First, carry addition may overflow and cause another carry. Second, this can ripple several limbs forward if multiple overflows happen sequentially.

Handling it correctly ensures the result is accurate for all inputs, regardless of content or size.

Comparison to 32-bit Versions

You might think that using 32-bit limbs on a 64-bit machine would simplify carry handling. But in practice, it’s a step backward:

- The number of limbs doubles for the same numeric value, increasing loop work.
- You still need to manage carries, just more of them.
- The compiler cannot optimize as aggressively with smaller types.
- You lose the advantage of CPU 64-bit mul and adc instructions.

As a result, using 64-bit limbs (or uintmax_t if you want portability) performs significantly better than using 32-bit limbs and later combining them.

Benefits of Linear Convolution

Linear convolution is the foundation of arbitrary-precision multiplication. Even though it has quadratic time complexity, it is simple to implement, exact, and easy to optimize for small input sizes.

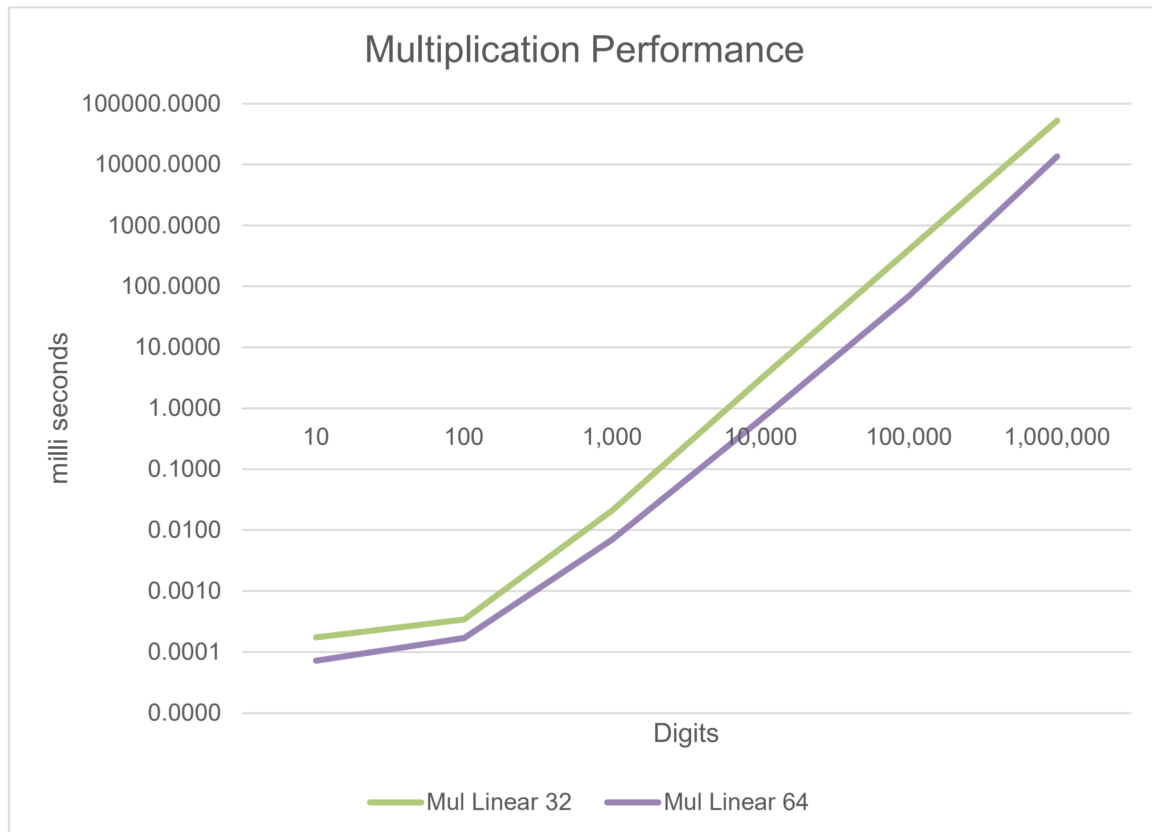
Key points:

- 64-bit limbs are optimal on 64-bit CPUs.
- Explicit carry handling avoids undefined behavior and ensures correctness.
- Carry must be fully propagated after each row of multiplication.
- Pre-sizing the result vector avoids reallocation.

- Avoid 32-bit decompositions unless necessary for special hardware or compatibility.

This method may not win speed races for large inputs, but it remains a reliable workhorse and the ideal base case for recursive multiplication strategies.

Performance of Linear Convolutions



As can be seen, the 64-bit linear convolution is faster (not surprisingly) than the 32-bit version.

Karatsuba multiplication

Invented in 1960 by A. Karatsuba. Before that, it was believed that it could not be faster than the schoolbook multiplication. Karatsuba showed that you can reduce the multiplication of two n-digit numbers to three multiplications and three additions/subtractions instead of the usual four multiplications.

Karatsuba's result was the first clear sign that long multiplication is not a fundamental limit. He began by writing each number in two roughly equal parts, say $x=x_1B^m+x_0$ and $y=y_1B^m+y_0$, where B is the limb base and is about half the digit-count. The standard expansion has four half-size products. x_1y_1, x_1y_0, x_0y_1, and x_0y_0, which together form xy.

Karatsuba noticed that only one extra product can recover the two cross terms. Compute

$$z_0=x_0y_0,\ z_2=x_1y_1,\ z_1=(x_1+x_0)(y_1+y_0).$$

The sum z_1 contains every cross contribution once, so

$$(x_1y_0+x_0y_1)=z_1-z_2-z_0.$$

With these three products and two linear-time add–subtract steps, the whole product is reconstructed as

$$xy=z_2B^{2m}+(z_1-z_2-z_0)B^m+z_0.$$

Replacing four half-size multiplies by three shifts the cost recurrence from $T(n)=4T(n/2)+O(n)$ to $T(n)=3T(n/2)+O(n)$. The solution drops from quadratic growth to $n^{\log_2 3} \sim n^{1.585}$. The savings at a single split are modest. Yet, when the process recurs, the benefit compounds, and once each half spans a few dozen machine limbs, Karatsuba outperforms schoolbook multiplication for multiplication with a higher number of decimals. This divide-and-conquer idea set the stage for every faster method that followed, from Toom–Cook to the FFT-based algorithms used in today's very large integer libraries.

Algorithm: Karatsuba multiplication

```
function karatsuba(a,b)
        if(a<=KARATSUBA_CUTOFF&& b<=KARATSUBA_CUTOFF)
                Return a*b      // Do multiplication of small numbers directly
        m=Numberofdigit(a)  // NumberofDigit() return the number of digits in base 10
        if(m>NumberofDigit(b))
                m=NumberofDigit(b)
        m=integer(m/2)

        // Splitting
        [ahigh,alow]=split(a,m)         // Split a into two half ahigh and alow
        [bhigh,blow]=split(b,m)         // Split b into two half bhigh and blow

        // Evaluation
        z0=karatsuba(alow,blow)
        z1=karatsuba(alow+ahigh,blow+bhigh)
        z2=karatsuba(ahigh,bhigh)

        // Recomposition
        return (z2*10^(2*m))+((z1-z2-z0)*10^m)+z0
```

Karatsuba algorithm reduces the complexity of multiplication from $O(n^2)$ to $O(n^{1.58})$

Karatsuba multiplication of two unsigned numbers

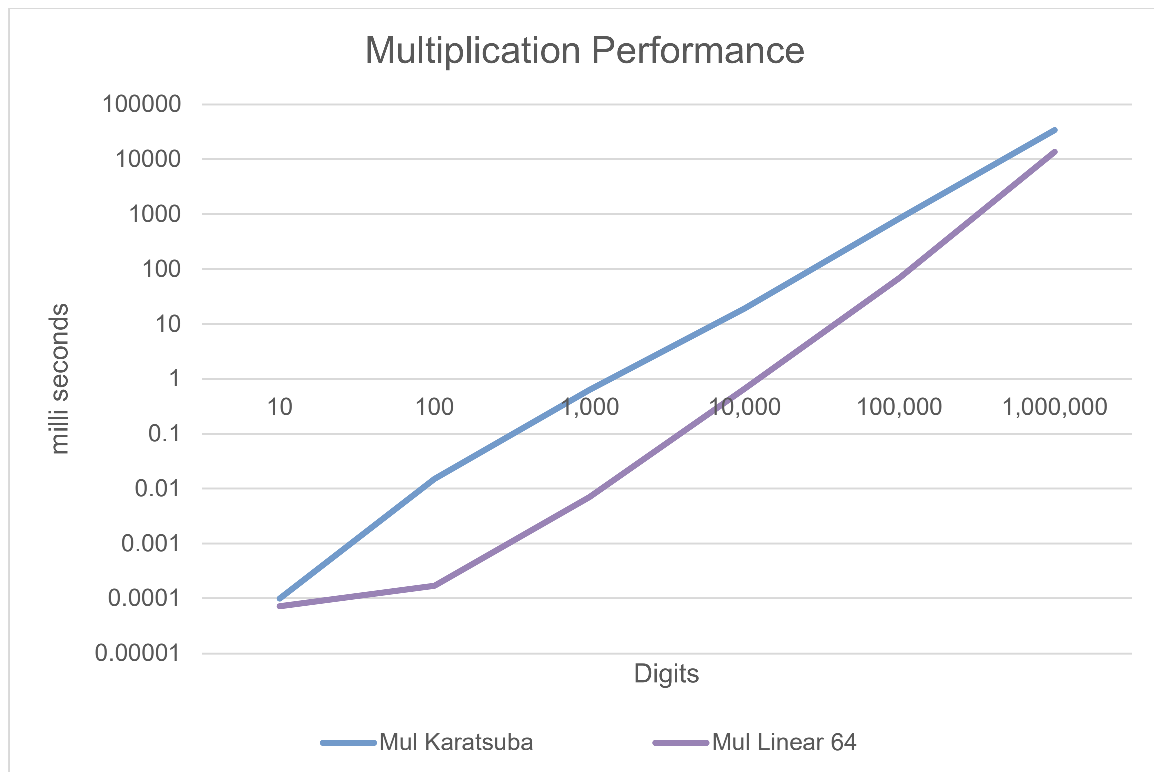
```
template<class _TY> inline mul128 _precision_umul64(const _TY a, const _TY b)
	{
	const _TY mask = 0xffffffff;
	const unsigned int shift = 32;
	const _TY a0 = a & mask, a1 = a >> shift;
	const _TY b0 = b & mask, b1 = b >> shift;
	const _TY a0b0 = a0*b0, a0b1 = a0*b1, a1b0 = a1*b0, a1b1 = a1*b1;
	_TY carry0, carry1, mid;
	struct mul128 res;
	mid = a0b1 + a1b0; carry1 = mid < a0b1 ? 1 : 0;
	res.lo = a0b0 + ( ( mid&mask ) << shift ); carry0 = res.lo < a0b0 ? 1 : 0;
```

```
	res.hi = (mid >> shift) + a1b1 + (carry1 << shift) + carry0; // no overflow
	return res;
	}

std::vector<iptype> karatsuba(const std::vector<iptype>& a,  const 
std::vector<iptype>& b)
{
	static constexpr std::size_t KARATSUBA_CUTOFF = 1; // tune for your CPU
	const std::size_t n = a.size();
	const std::size_t m = b.size();
	int wrap;

	//-----  short operands -> plain O(n²) multiply  ----------------
	if (n == 1 && m == 1)  // If max digits in lhs & rhs less than to fit into 
a 64 bit integer then do it the binary way
	{
		mul128 tmp = _precision_umul64(a.front(), b.front());
		std::vector<iptype> result;
		result.push_back(tmp.lo);
		result.push_back(tmp.hi);
		_int_precision_strip_trailing_zeros(result);
		return result;
	}

	//-----  quick exits for 0 and 1  ------------------------------
	if (n == 1 && a[0] == 0) return { 0 };
	if (m == 1 && b[0] == 0) return { 0 };
	if (n == 1 && a[0] == 1) return b;
	if (m == 1 && b[0] == 1) return a;

	if (n <= KARATSUBA_CUTOFF && m <= KARATSUBA_CUTOFF)
	{
		// Use linear multiplication for small operands
		return _int_precision_umul_linear(a, b);
	}

	/*-----  split -------------------------------------------------*/
	const std::size_t k = (std::max(n, m) + 1) >> 1;  // ⌈max/2⌉
	// low part = first k limbs (or fewer if vector is short)
	// high part = the rest (may be empty)

	auto split = [k](const std::vector<iptype>& v)
		{
			std::vector<iptype> low(v.begin(), v.begin() + std::min(k, 
v.size()));
			std::vector<iptype> high(v.begin() + std::min(k, v.size()), 
v.end());
			if (high.empty()) high.push_back(0);  // keep “high” non-
empty
			return std::pair<std::vector<iptype>, 
std::vector<iptype>>(std::move(low),
				std::move(high));
		};

	auto [al, ah] = split(a);
	auto [bl, bh] = split(b);

	//-----  three recursive products ------------------------------
	std::vector<iptype> z0 = karatsuba(al, bl);		// P0  = al·bl
	std::vector<iptype> z2 = karatsuba(ah, bh);		// P2  = ah·bh
```

```
	std::vector<iptype> al_plus_ah = _int_precision_uadd(al, ah);
	std::vector<iptype> bl_plus_bh = _int_precision_uadd(bl, bh);

	std::vector<iptype> z1 = karatsuba(al_plus_ah, bl_plus_bh); // P1
	std::vector<iptype> t = _int_precision_uadd(z0, z2);
	std::vector<iptype> z1_minus = _int_precision_usub(&wrap, z1, t);  // P1 − 
P0 − P2

	//-----  assemble result ------------------------------------
	z2 = _int_precision_ushiftleft(z2, 2 * k * Bitsiptype);   // P2 · R²ᵏ
	z1_minus = _int_precision_ushiftleft(z1_minus, k * Bitsiptype); // (P1−...) 
· Rᵏ

	std::vector<iptype> res = _int_precision_uadd(z2, z1_minus);
	res = _int_precision_uadd(res, z0);
	return res;
}
```

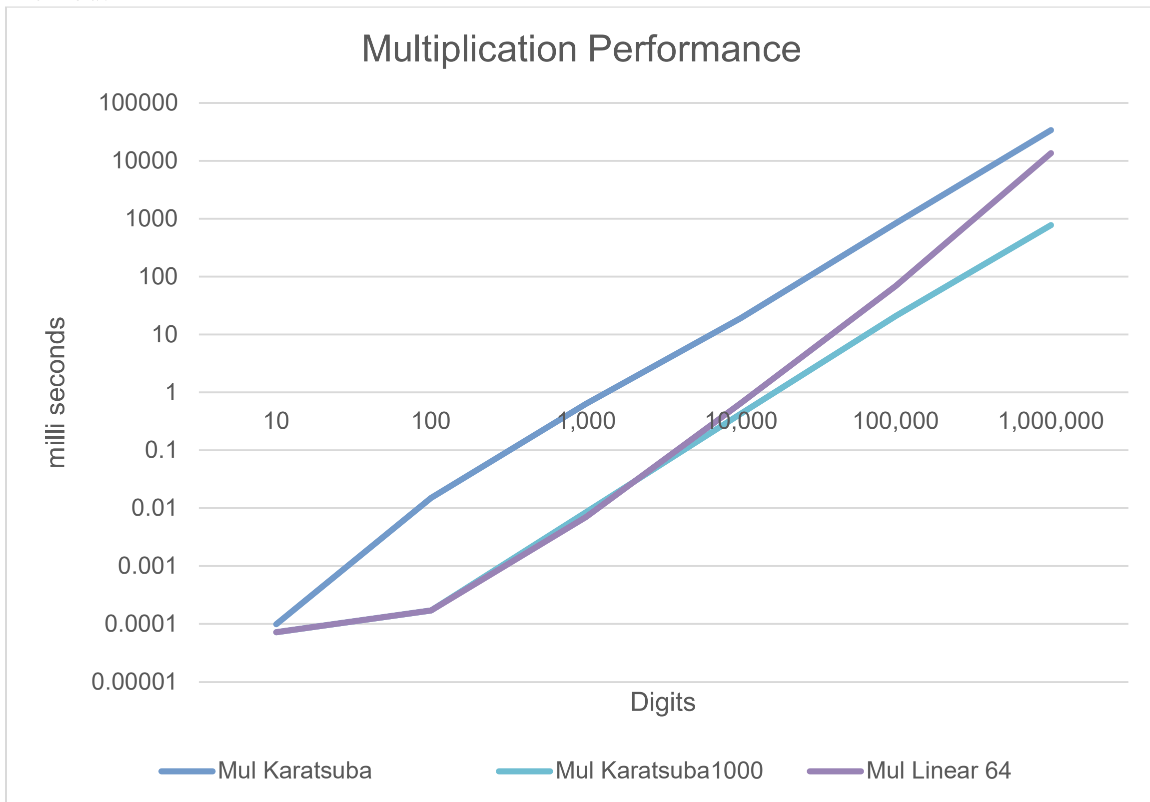
We notice that we continue to split until we are down to two single iptype (same as uintmax_t). This approach is not the most efficient, as the split and recursion will begin to impact performance negatively. Instead, you can choose a threshold (KARATSUBA_CUT_OFF) from which you can call the linear convolution multiplication for better performance.

Performance of Karatsuba



Applying a threshold (around 1,000 decimal digits) where you stop Karatsuba recursion and call the multiply linear convolution, you get a more realistic and much better performance out of Karatsuba. See below. The Karatsuba method accelerates away from

the linear convolution method above 1,000 decimal digits. The actual threshold depends on the underlying architecture and the implementation of the arbitrary precision library. This threshold needs to be fine-tuned per implementation of the arbitrary precision method.



Using a cut-off threshold around 1,000 decimal digits (Mul Karatsuba1000), you can exploit the Karatsuba method and see that it is significantly faster than the linear convolution above the 1,000 decimal mark.

Toom-Cook multiplication

The Toom-Cook algorithm was invented in 1963 by A. Toom and S. Cook. Instead of splitting the number into two halves as used by Karatsuba, they could split it into any number k, however, with increasing complexity. Karatsuba algorithm is equivalent to k=2 and named Toom-Cook-2. The most common variation is splitting the number into three parts (Toom-Cook-3); however, GNU arbitrary precision also offers four-part splitting (Toom-Cook-4).

The complexity of Toom-Cook-3 is $O(n^{1.46})$ and Toom-Cook-4 is $O(n^{1.40})$

Algorithm: Toom-Cook-3

```
function toomcook3(a,b)
        if(a<=TC3_CUTOFF|&&b<= TC3_CUTOFF)
                return a*b        // Do multiplication of small numbers directly
        m=Numberofdigit(a)   // NumberofDigit() return the number of digits in base 10
```

```
    if(m>NumberofDigit(b))
        m=NumberofDigit(b)
    m=integer(m/3)

    // Splitting
    // Split a into three half ahigh, amid and alow
    [ahigh,amid,alow]=split3(a,m)
     // Split b into three half bhigh, bmid, and blow
    [bhigh,bmid,blow]=split3(b,m)

    // Evaluation
    p1=alow+ahigh+amid
    p2=alow+ahigh-amid
    p3=2(p2+ahigh)-alow
    q1=blow+bhigh+bmid
    q2=blow+bhigh-bmid
    q3=2(q2+bhigh)-blow

    // Pointwise multiplication
    i0=toomcook3(alow,blow)
    i1=toomcook3(p1,q1)
    i2=toomcook3(p2,q2)
    i3=toomcook3(p3,q3)
    i4=toomcook3(amid,bmid)

    // Interpolation
    i3=(i3-i1)/3
    i1=(i1-i2)/2
    i2=-i0
    i3=(i2-i3)/2+2*i4
    i2=i2+i1-i4
    i1=i1-i3

    // Recomposition
    I1=i1*10^m
    I2=i2*10^2m
    I3=i3*10^3m
    I4=i4*10^4m
    result=i0+i1+i2+i3+i4
    if(sign(a)*sign(b)<0)
        result=-result
    return result
```

Toom-cook three-way splitting code for multiplication of two numbers.

```
int_precision toomcook3(const int_precision& lhs, const int_precision& rhs)
{
	static constexpr std::size_t TOOM3_CUTOFF = 1;
```

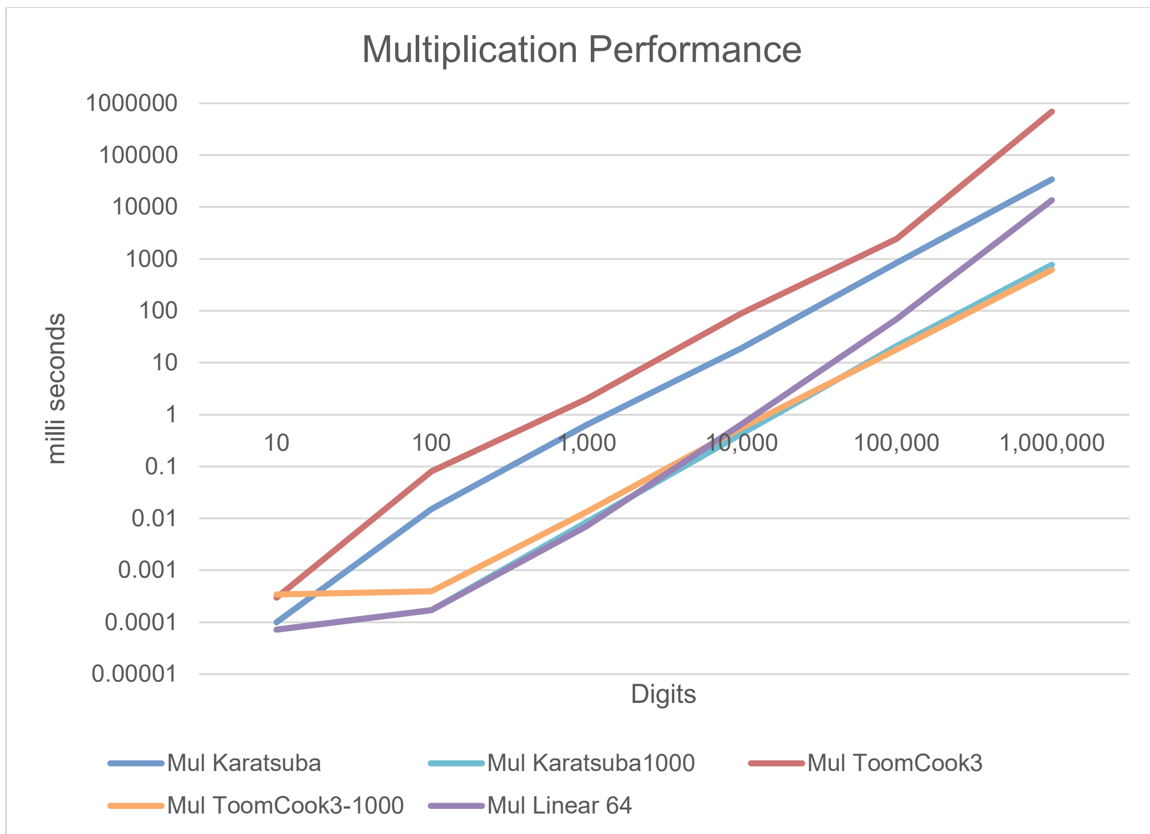
```
	static const int_precision  TWO(2);
	static const int_precision  THREE(3);
	const auto& L = lhs.numberRef();
	const auto& R = rhs.numberRef();
	const std::size_t nL = L.size();
	const std::size_t nR = R.size();
	const int_precision c2(2);
	int_precision res(0);

	// Base cases
	if (nL == 1 && lhs.iszero() || nR == 1 && rhs.iszero())//lhs * 0 or rhs*0
is zero
		return res;		// Return zero
	if (nL == 1 && lhs.index(0) == 1)//rhs * |1| is rhs
	{
		res = rhs;
		res.sign(lhs.sign() * rhs.sign());
		return res;
	}
	if (nR == 1 && rhs.index(0) == 1)//lhs * |1| is lhs
	{
		res =lhs;
		res.sign(lhs.sign() * rhs.sign());
		return res;
	}

	if (nL <= TOOM3_CUTOFF && nR <= TOOM3_CUTOFF) {
		res = lhs * rhs;
		return res;
	}

	// helpers local to toomcook3new() ------------------------------------
	auto shl_limbs = [](int_precision& v, std::size_t limbs)
		{
			if (limbs == 0 || v == 0) return;
			v.pointer()->insert(v.pointer()->begin(), limbs, 0);
		};

	auto make_slice = [](const std::vector<iptype>& vec, std::size_t from,
std::size_t count) -> int_precision
		{
			if (from >= vec.size()) return int_precision(0);

			std::size_t take = std::min(count, vec.size() - from);
			return int_precision(std::vector<iptype>(vec.begin() + from,
vec.begin() + from + take));
		};

	// 1. split ----------------------------------------------------------------
	const std::size_t k = (std::max(nL, nR) + 2) / 3;  // ceil(max/3)

	int_precision a0 = make_slice(L, 0, k);
	int_precision a1 = make_slice(L, k, k);
	int_precision a2 = make_slice(L, 2 * k, k);		// may be < k limbs
	int_precision b0 = make_slice(R, 0, k);
	int_precision b1 = make_slice(R, k, k);
	int_precision b2 = make_slice(R, 2 * k, k);

	// 2. evaluation -----------------------------------------------------------
	int_precision s0 = a0 + a2;		// a0 + a2
	int_precision t0 = b0 + b2;		// b0 + b2
```

```
        int_precision a1p = s0 + a1;           // a( 1)
        int_precision b1p = t0 + b1;           // b( 1)
        int_precision am1 = s0 - a1;           // a(-1)
        int_precision bm1 = t0 - b1;           // b(-1)
        /*  IMPORTANT: shift by literal 1 bit, not by int_precision(1)  */
        int_precision am2 = ((am1 + a2) * c2) - a0;   // a(-2)
        int_precision bm2 = ((bm1 + b2) * c2) - b0;   // b(-2)

        // 3. pointwise products -----------------------------------------------
        int_precision w0 = toomcook3(a0, b0);
        int_precision w4 = toomcook3(a2, b2);
        int_precision w1 = toomcook3(a1p, b1p);
        int_precision w2 = toomcook3(am1, bm1);
        int_precision w3 = toomcook3(am2, bm2);

        // ---------- 4. classic interpolation (all non-negative) -------
        // r0 = w0, r4 = w4 already done
        int_precision r1 = (w1 - w2) / TWO;              // (w1 ? w2)/2
        int_precision r2 = w2 - w0;                      // w2 ? w0
        int_precision r3 = (w3 - w1) / THREE;            // (w3 ? w1)/3

        r3 = (r2 - r3) / TWO + (w4 * TWO);               // final r3
        r2 = r2 + r1 - w4;                               // final r2
        r1 = r1 - r3;                                    // final r1
        // now r0..r4 are the degree-4 coefficients

        // ---------- 5. recomposition --------------------------------
        shl_limbs(r1, k);
        shl_limbs(r2, 2 * k);
        shl_limbs(r3, 3 * k);
        shl_limbs(w4, 4 * k);

        res = w0 + r1;
        res += r2;
        res += r3;
        res += w4;
        res.sign(lhs.sign() * rhs.sign());
        return res;
}
```

Notice that we are working with signed arbitrary precision integers (int_precision) and not just vectors of unsigned limbs. Since we subtract values, we need to handle it as signed subtraction to maintain a check on the signed value, even when the two input variables are both positive numbers.

Performance of ToomCook3

Applying a cut-off threshold for ToomCook3, as we did for Karatsuba, results in a tremendous performance gain. As expected, ToomCook3 surpasses Karatsuba by around 100,000 decimal digits. Again, we must mention that the recommended cut-off threshold depends on implementing the arbitrary precision library.



Fast Fourier Transformation (FFT)

It is beyond the scope of this paper to explain the theory behind FFT multiplication, but readers can reference [4] for more information.
However, the method consists of converting the two numbers an and bm via a series of Fourier transformations, multiplying them together, and then doing the inverse Fourier transformation of the result back to the digital domain.
FFT complexity is O(n·log(n)), making it preferable over the other multiplication methods.

Algorithm: FFT multiplication

```
function multiplyFFTRadix2(A, B, chunkBits):
   // 1. Validate chunk size
   if chunkBits ∉ {4, 8, 16}:
      error "chunkBits must be 4, 8, or 16"

   // 2. Unpack limbs into “digits” (base = 2^chunkBits)
   fa ← unpack(A, chunkBits)   // returns complex vector of length lenA
   fb ← unpack(B, chunkBits)   // returns complex vector of length lenB
```

```
   // 3. Choose FFT length N = next power of two ≥ (lenA + lenB − 1), then double for safety
   need ← len(fa) + len(fb) − 1
   N ← 1
   while N < need:
      N ← N << 1
   N ← N << 1
   resize fa and fb to N (zero-pad)

   // 4. Forward FFT on both
   fft2(fa, invert = false)
   fft2(fb, invert = false)

   // 5. Pointwise multiply
   for i in 0..N−1:
      fa[i] ← fa[i] * fb[i]

   // 6. Inverse FFT and scale
   fft2(fa, invert = true)   // divides by N

   // 7. Repack digits and propagate carries
   result ← packAndCarry(fa, chunkBits)

   // 8. Trim leading zero-limbs
   while size(result) > 1 and last(result) == 0:
      remove last(result)

   return result
```

Note: the FFT2 function is the Fast Fourier Transform with Radix 2.

Fast Fourier Transformation using radix 2.

```
// Iterative in-place radix-2 FFT with integrated bit reversal
void fft2(std::vector<std::complex<double>>& a, bool invert) {

	size_t n = a.size();
	if (n == 0 || (n & (n - 1)) != 0) {
		throw std::invalid_argument("Size of input must be a power of two.");
	}
	constexpr double PI = 3.14159265358979323846264;

	// Bit-reversal permutation
	size_t log2n = 0;
	while ((1ULL << log2n) < n) ++log2n;

	for (size_t i = 0; i < n; ++i) {
		size_t rev = 0;
		for (size_t j = 0; j < log2n; ++j) {
			if ((i >> j) & 1) {
				rev |= 1ULL << (log2n - 1 - j);
			}
		}
```

```
		if (i < rev) {
			std::swap(a[i], a[rev]);
		}
	}

	// Iterative FFT
	for (size_t len = 2; len <= n; len <<= 1) {
		double angle = 2 * PI / len * (invert ? -1 : 1);
		std::complex<double> wlen(std::cos(angle), std::sin(angle));
		for (size_t i = 0; i < n; i += len) {
			std::complex<double> w(1);
			for (size_t j = 0; j < len / 2; ++j) {
				std::complex<double> u = a[i + j];
				std::complex<double> v = a[i + j + len / 2] * w;
				a[i + j] = u + v;
				a[i + j + len / 2] = u - v;
				w *= wlen;
			}
		}
	}

	// If inverse FFT, divide by n
	if (invert) {
		const double inv = 1.0 / n;
		for (std::complex<double>& x : a)
			x *= inv;
	}
}

vector<iptype> multiplyFFTRadix2(const vector<iptype>& A, const vector<iptype>&
B, const int chunkBits = 8)
{
	if (chunkBits != 4 && chunkBits != 8 && chunkBits != 16)
		throw std::invalid_argument("chunkBits must be 4, 8, or 16");

	const int base = 1 << chunkBits;
	const int chunksPerLimb = sizeof(iptype) * 8 / chunkBits;
	const uint64_t mask = base - 1;

	// Estimate the number of chunks and pre-allocate fa and fb
	vector<complex<double>> fa(A.size() * chunksPerLimb * 2);
	vector<complex<double>> fb(B.size() * chunksPerLimb * 2);

	// Preliminary byte lengths after 8-bit splitting
	size_t lenA = A.size();
	size_t lenB = B.size();
	size_t need = lenA + lenB - 1;        // linear-conv length

	// N = power-of-2 >= need, then ×2 once more (safety)
	size_t N = 1;  while (N < need) N <<= 1;
	N <<= 1;

	// 1.  load limbs -> FFT buffers  (little-endian)  and count bytes

	lenA = 0;                   // number of non-zero chunksPerLimb in A
	for (iptype limb : A) {     // limb 0 first (least-sig limb)
		for (int i = 0; i < chunksPerLimb; ++i) {
			iptype chunk = (limb >> (i * chunkBits)) & mask;
			fa[lenA++] = complex<double>(static_cast<double>(chunk),
0.0);
		}
```

```
	}
	// trim trailing high zeros that belong to the MS side
	while (lenA > 1 && fa[lenA - 1].real() == 0.0) --lenA;

	lenB = 0;                   // number of non-zero chunksPerLimb in B
	for (iptype limb : B) {     // limb 0 first (least-sig limb)
		for (int i = 0; i < chunksPerLimb; ++i) {
			iptype chunk = (limb >> (i * chunkBits)) & mask;
			fb[lenB++] = complex<double>(static_cast<double>(chunk), 
0.0);
		}
	}
	// trim trailing high zeros that belong to the MS side
	while (lenB > 1 && fb[lenB - 1].real() == 0.0) --lenB;
	// Now len A and LenB is the actual number of chunksPerLimb (16, 8 or 4-
bits)

	// 2.  recalculate transform length
	need = lenA + lenB;                     // one extra byte for final carry
	N = 1;
	while (N < need - 1) N <<= 1;           // first 4^k >= lenA+lenB-1
	N <<= 1;                              // extra safety stage
	fa.resize(N);  fb.resize(N);          // zero-pad to N

	// 3. FFT - point-wise multiply - inverse FFT
	fft2(fa, false);
	fft2(fb, false);
	for (size_t i = 0; i < N; ++i) fa[i] *= fb[i];
	fft2(fa, true);                     // divides by N - scale is correct

	// Step 4 & 5: Combined carry propagation and limb construction
	vector<iptype> result;
	result.reserve((need + chunksPerLimb - 1) / chunksPerLimb);

	iptype carry = 0;
	iptype limb = 0;
	int bitsFilled = 0;

	for (size_t i = 0; i < need; ++i) {
		iptype value = static_cast<iptype>(fa[i].real() + 0.5) + carry;
		iptype chunk = value & mask;
		carry = value >> chunkBits;

		limb |= (chunk << bitsFilled);
		bitsFilled += chunkBits;

		if (bitsFilled >= 64) {
			result.push_back(limb);
			limb = 0;
			bitsFilled = 0;
		}
	}

	if (bitsFilled > 0)
		result.push_back(limb);
	else if (carry)
		result.push_back(carry);

	// Trim leading zeros
	while (result.size() > 1 && result.back() == 0)
		result.pop_back();
```

```
    return result;                        // res[0] = least-significant limb
}
```

However, looking at the algorithm, we can add some relevant comments. We can break down the FFT-based multiplication process into four steps. The goal is to make the concepts accessible, intuitive, and connected to the implementation of arbitrary-precision multiplication.

Step 1. Unpacking a and b into fa and fb.

In the unpacking step, you convert the vector of limbs A into a vector<complex<double>> fa. Now, as will become clear in the section on the limitations of FTT, you need to unpack into a sequence of bytes, meaning that each input limb (64-bit integers) will be unpacked into eight entries in the vector<complex<double>> fa. This is needed to prevent inaccuracies from appearing in our multiplication result, as described in a later sub-chapter.

Step 2. What happens in fa = FFT(a) and fb = FFT(b)?

The FFT (Fast Fourier Transform) converts the input from its original form, typically a list of digits or coefficients, into the frequency domain. But what does that really mean?

Let's back up a bit. We interpret a large integer (or a std::vector<uint64_t>) as a polynomial where each limb is a coefficient:

$$A(x)=a_0+a_1x+a_2x^2+\cdots+a_{n-1}x^{n-1}$$

This is not a random trick; it maps perfectly onto how base-2^{64} representations work. Now, if we multiply two such integers, it becomes a polynomial product:

$$C(x)=A(x)\cdot B(x)$$

To multiply two polynomials using the FFT, we evaluate each polynomial at several fixed points (roots of unity), multiply the values point by point, and then use an inverse transform to convert back to the coefficient form.

This process is precisely what happens when you do:

fa = fft2(fa)
fb = fft2(fb)

Here:

- fa and fb are vectors of complex<double> coefficients.
- FFT(fa) gives you a new vector of complex numbers.
- These values represent the evaluations of the input polynomial at various points evenly spaced around the complex unit circle.

This is what's meant by transforming into the frequency domain. It converts convolution (the sum of products) into pointwise multiplication, which is significantly faster.

Why do we need the bit-reversal before the main Cooley–Tukey loops?

You'll see this "bit-reversal" step:

Bit reversal code snippet.

```
// Bit-reversal permutation
size_t log2n = 0;
while ((1ULL << log2n) < n) ++log2n;

for (size_t i = 0; i < n; ++i) {
    size_t rev = 0;
    for (size_t j = 0; j < log2n; ++j) {
        if ((i >> j) & 1) {
            rev |= 1ULL << (log2n - 1 - j);
        }
    }
    if (i < rev) {
        std::swap(a[i], a[rev]);
    }
}
```

It reorders each element at index *i* into position *rev*, where *rev* is just the bits of *i* flipped end-for-end. After this pass, the array is in "bit-reversed" order.

The radix-2 FFT works by repeatedly grouping elements into pairs whose indices differ in exactly one bit (the "butterfly" operations). In our iterative in-place version, you want to:

- Do all length-2 butterflies first (stride = 1),
- Then all length-4 butterflies (stride = 2),
- Then length-8 (stride = 4), and so on.

If the data isn't in bit-reversed order up front, those butterflies will hop around all over memory, and you'd need extra indexing logic (or temporary buffers) at each stage to find the correct pairs. By performing the bit-reversal ahead of time, each successive pass can take contiguous blocks of size *len* and split them in half without the need for complex indexing within the inner loops.

There are at least three performance benefits of this.

No extra copies are required during each FFT pass, since the data is already arranged so that each butterfly "butterflies" two neighbors. It is more cache-friendly inner loops. After bit-reversal, the memory accesses in the following passes are sequential, which modern CPUs love. And finally, it results in simpler code: the only nontrivial reordering happens once; the core FFT loops stay tight and branch-predictable.

Step 3. Why can we multiply fa[i] * fb[i] pointwise?

This is the big trick of the FFT.

Polynomial multiplication requires computing:

$$c_k = \sum_{i+j=k} a_i \cdot b_j$$

That is a convolution: for every position k, we must multiply and sum all pairs of terms whose indices add up to k. This takes $O(n^2)$ time directly.

But if we evaluate both polynomials at the same set of points, say:

$fa=[A(w^0),A(w^1),\dots,A(^{wn-1})]$
$fb=[B(w^0),B(w^1),\dots,B(w^{n-1})]$

Then pointwise multiplication gives the result of their product at the same points:

$fc[i]=fa[i]\cdot fb[i]=A(w^i)\cdot B(w^i)=C(w^i)$

That means we now have the values of the product polynomial already evaluated at all points w^i, and there is no need to do any complicated convolution manually, multiply the values element by element with no carry, shifting, or nested loops.

This is the key advantage that the FFT turns a slow convolution into pointwise multiplication, which is much faster.

Step 4. What does the inverse FFT(fc) do? And why carry propagation?

After we have all the pointwise products:

fc[i] = fa[i]·fb[i]

We now need to return the actual coefficients of the result polynomial, as well as the vector of limbs that make up the integer product.

This is what the inverse FFT does:

result = FFT(fc, true); // The true indicate the reverse transformation

This step reconstructs the coefficients $c_0,c_1,\dots,c_{n+m-2}$ from the evaluated points. These are the raw coefficients of the product polynomial.

But there's a catch:

- These coefficients are not guaranteed to be under 2^{64}
- They are just floating-point approximations due to complex arithmetic in FFT()
- So, they can be larger than a limb and do not account for carries.

Carry propagation

The final step is to normalize these coefficients back into proper 64-bit limbs. This is where carry propagation happens:

```
for (int i = 0; i < result.size(); ++i) {
    result[i+1] += result[i] >> 64;
    result[i] &= 0xFFFFFFFFFFFFFFFF;
}
```

Or the equivalent in 128-bit arithmetic. This ensures that:

- Each limb is under 2^{64}.
- The overflow from each limb is added to the next one.
- You end up with the correct binary representation of the whole product.

Without this step, you would get an "almost correct" result; the coefficients would be close, but the digits wouldn't match what you'd get from normal multiplication.

Step	What it does
FFT(a)	Evaluates the polynomial A(x) at fixed points (frequency domain)
FFT(b)	Same for B(x)
Multiply	Computes $A(w^i) * B(w^i) = C(w^i)$ at all points, no carry needed
Inverse FFT()	Converts evaluations of C(x) back to raw coefficients
Normalize	Propagate carry and reduce each coefficient to 64-bit limbs (base 2^{64})

Why does FFT radix-2 require a power-of-two input length?

The classic radix-2 FFT (Cooley-Tukey algorithm) works by repeatedly splitting the input array into even and odd parts:

$$FFT(x)=FFT(\text{even terms})+w\cdot FFT(\text{odd terms})$$

Each split reduces the problem's size by half, a process that occurs recursively.

This binary splitting process only works cleanly if the array size is exactly a power of 2. For example:

- 8 → splits to 4 and 4 → splits to 2 and 2 → splits to 1 and 1 (base case).
- But 10 → splits to 5 and 5 → can't split evenly again, so the algorithm breaks.

So, for the recursion to reach base cases (size one arrays) evenly, the full length must be 2^n.

FFT relies on evaluating the input at specific roots of unity points, like:

$$w_k=e^{-2\pi ik/N}, \text{ for } k=0,1,\ldots,N-1$$

Where N is the input length.

To perform the transform correctly, these roots must satisfy special periodic properties like:

$$w_{k+N/2} = -w_k$$

These properties only hold cleanly when N is a power of 2. The whole butterfly structure of the FFT, where data is mixed using these roots, depends on these exact symmetries.

If N is not a power of 2, then the symmetry breaks, and the roots don't line up as the algorithm expects, resulting in incorrect results.

When we pad the input with zeros, we're extending the polynomial with terms that are zero. This doesn't change the value of the integer it represents.

For example:

- If a = [5, 2] represents 5+2x, then padding it to 4 terms gives $5+2x+0x^2+0x^3$.

Mathematically, the result remains valid. You ensure the FFT has the correct size to operate.

Reason	Why does FFT need a power-of-2 length
Algorithm structure	Radix-2 FFT splits the array evenly, recursively, which only works with sizes like 2^n
Roots of unity	Require exact symmetry, which only holds when the size is 2^n
Padding is safe	Zero-padding extends the number without altering its value, enabling the FFT to operate correctly.

Why do we require input size ≥ 2^n, and then double it again (n <<= 1)?

Say you are multiplying two vectors a and b of sizes n and m, respectively. The product will have at most n + m - 1 non-zero terms.

So, your FFT must be able to represent up to n + m - 1 coefficients after convolution.

To do that using radix-2 FFT, you need to choose a size N that is the smallest power of two such that:

$$N \geq n+m-1$$

Then you zero-pad both a and b to size N.

It is a common trick to double the input size after the initial requirement of input size ≥ 2^n done for two main reasons:

a) You start with n = max(a.size(), b.size()), then do n <<= 1

This guarantees:

$$n \geq 2 \cdot \max(a,b) \geq n+m-1$$

This simplifies the allocation logic and avoids the need to compute n + m - 1, followed by the next power of two.

It's safe, slightly conservative, and ensures enough space.

b) Some implementations double it to avoid wrap-around issues

Specific convolution kernels (especially circular ones) can wrap around if the size is too small. Doubling to 2N ensures enough zero-padding to avoid overlap or aliasing. This is especially true in modular convolution or when using FFT over finite fields.

Is it strictly needed? No, not always.

If you precisely calculate the required FFT size as:

size_t size = next_power_of_two(a.size() + b.size() - 1);

And pad both inputs to that size, then you don't need to double it further.

But doing:

```
size_t size = 1;
while (size < max(a.size(), b.size())) size <<= 1;
size <<= 1; // extra doubling
```

It is often done to ensure safety and simplicity, and to account for:

- Rounding behavior in FFT (especially floating-point precision).
- Oversized buffers for in-place operations.
- Preallocated workspaces or SIMD batch size alignment.

Step	Reason
n + m - 1	Minimum number of coefficients in the result
next_power_of_two()	Needed for radix-2 FFT
n <<= 1 (extra doubling)	Conservative sizing, avoids wrap-around, simplifies code

You can skip the extra n <<= 1 if you carefully compute next_power_of_two(n + m - 1), but the doubling ensures robustness and is a standard safe default in many FFT libraries.

Can you stop carrying propagation after n + m - 1 digits?

Yes, in theory. After performing the Inverse FFT, you get the coefficients of the result polynomial. The actual result of multiplying n and m-limb integers can never require more than n + m limbs, more precisely: n + m - 1 meaningful coefficients, and possibly one extra carry limb. So, you only need to normalize and propagate carry across the first n + m limbs.

Any limbs beyond that are pure zero (if inputs were padded correctly), or just numerical garbage caused by padding, should be discarded.

Limitations of the FFT

FFT uses floating-point arithmetic using the double type in C++. In the initial step, you need to map the vector of 64-bit binary digits into a vector of complex< doubles>. You do that by splitting each 64-bit binary digit into eight bytes and then converting each byte

into a complex<double>. Because we use floating-point arithmetic in FFT, we need to be careful with how large the two numbers we multiply can be. In [4], they give a formula for the number of decimal digits *n* that the FFT can handle without inaccuracy in the result as a function of the initial splitting into bytes and how large our double mantissa is in bits (53 bits in a C++ double)

$$\log_2((\text{Sample size})^2) + \log_2(n) + k \cdot \log_2(\log_2(n)) < 53 \quad (1)$$

For example, a byte (8-bit) has a sample size of $2^8 = 256$. Where k is "a few". Let's choose k = 2. We get that for n = 100,000,000 (100 million), 52 < 53, which is true.

If we solve the above equation for *n*, we get an *n* of approximately 175M digits. Unfortunately, k as two is insufficient, as we encounter random errors in the multiplication result when multiplying by 125M+ digits. Instead, I recommend using k as a factor of 2.15, which yields a maximum multiplication size limit of approximately 116 million digits.
That leads to the question of what to do if you need more digits in multiplication. A simple solution is to reduce the sample size to 4 bits, rather than 8 bits. Using the above formula again, you can multiply a maximum of 17.8 billion digits. If that is still insufficient, you can do a 2-bit sample size and get a limitation of 229 billion digits. Every time you halve the sample size, the memory requirements increase by a factor of four, thereby reducing the overall performance of FFT multiplication. If you go the other way and use a sample size of 16 bits, you get a limitation of approximately 8,396 digits, which is excessively small to be useful in practice.

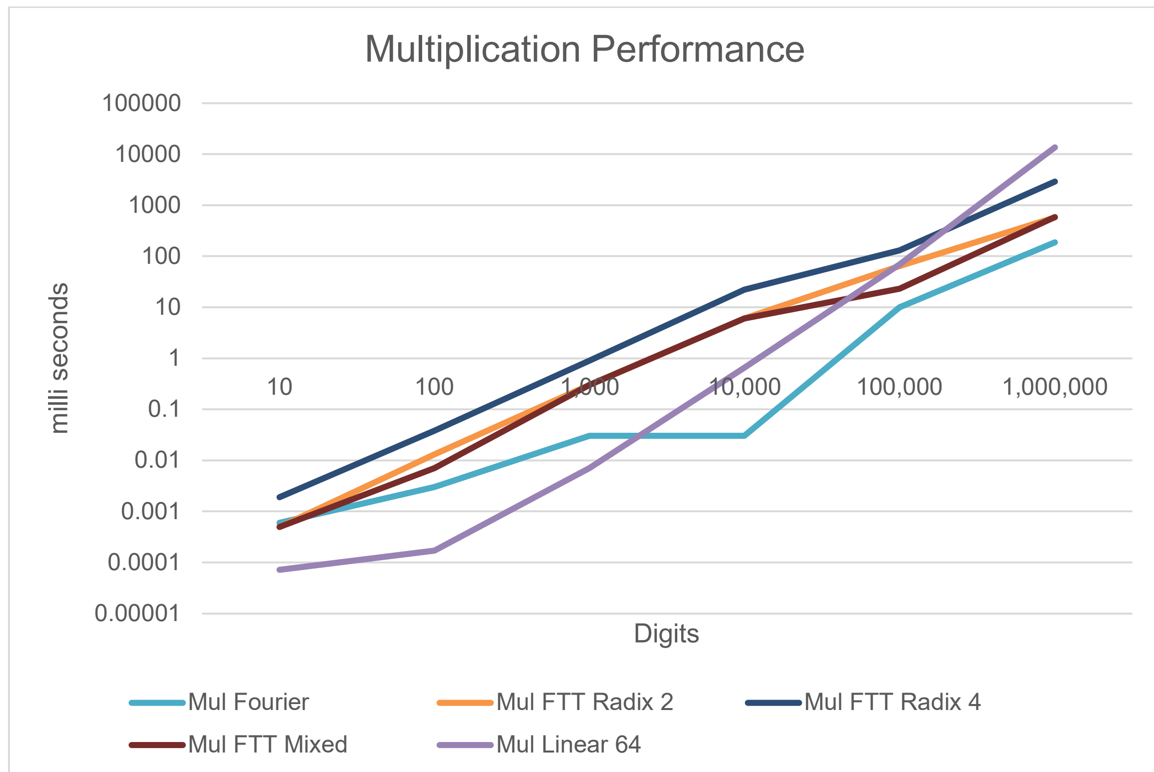
The limitations can vary depending on which system you are running and if it supports floating-point arithmetic above 64 bits. IEEE 754 specifies an extended 80-bit version, sometimes referred to as long double. (Not all compiler supports it, so the author's arbitrary precision packages only support the standard 64-bit double)

Maximum operand size for multiplication as a function of sample size in the FFT

C++ type	Double	Long double*
Bits in Mantissa	53-bit	64-bit
Sample size		
16-bit	8,396	5.3M
8-bit	116M	120B
4-bit	17.8B	20T
2-bit	229B	280T

*) Not supported on all compilers and systems
Notice that a 32-bit sample will overflow the double 53-bit mantissa

Performance of FTT



The specialized Mul Fourier is the fastest version (don't use the complex<double> library). The other version was not designed for speed, but for clarity; as expected, it is slower. The Mul FFT Mixed determines automatically if the FTT radix two or radix four is the preferred code to run based on the size of the multiplying operands. As a reference, the multiplication using 64-bit linear convolution is also shown.

NTT method.

The Number Theoretic Transform (NTT) is the modular arithmetic counterpart of the Fast Fourier Transform (FFT). While the FFT works over the complex numbers, the NTT performs similar convolution operations using only integers modulo a prime. The technique builds on concepts from number theory, such as primitive roots of unity modulo a prime. It is ideal for applications that require exact arithmetic, particularly in cryptography and arbitrary-precision multiplication.

The first notable uses of NTT-like techniques in convolution go back to the 1960s and 1970s, when the Cooley–Tukey FFT gained prominence. Over time, NTT gained popularity in contexts where floating-point roundoff errors could not be tolerated, such as in Cryptographic schemes (RSA, lattice cryptography), Modular polynomial

multiplication, and large integer multiplication in libraries like FLINT or implementations of Schönhage–Strassen variants.

Modern usage often involves combining multiple NTTs over different primes and reconstructing the final result via the Chinese Remainder Theorem (CRT).

Algorithm for the NTT 32-bit version using either 3 or 6 primes.

```
function multiplyNTT32(A, B, useSixPrimes = false):
   // 1. Split 64-bit limbs into 16-bit chunks (little-endian)
   aC ← unpackToChunks(A, 16)   // returns vector<uint32_t>
   bC ← unpackToChunks(B, 16)

   // 2. Choose transform length N = next power-of-2 ≥ (len(aC) + len(bC))
   need ← len(aC) + len(bC)
   N ← 1
   while N < need:
      N ← N << 1

   // 3. Perform NTT convolution under each prime
   //    – the first three primes are always used
   r0 ← nttConvolution(aC, bC, primeIndex = 0, N)
   r1 ← nttConvolution(aC, bC, primeIndex = 1, N)
   r2 ← nttConvolution(aC, bC, primeIndex = 2, N)

   if useSixPrimes:
      r3 ← nttConvolution(aC, bC, primeIndex = 3, N)
      r4 ← nttConvolution(aC, bC, primeIndex = 4, N)
      r5 ← nttConvolution(aC, bC, primeIndex = 5, N)

   // 4. Combine residues via CRT
   if useSixPrimes:
      // pairwise CRT to form three 62-bit residues
      c0 ← crtCombine2(r0, r1, primes 0&1)
      c1 ← crtCombine2(r2, r3, primes 2&3)
      c2 ← crtCombine2(r4, r5, primes 4&5)
      // three-way CRT merge into 64-bit coefficients
      coeff ← crtCombine64(c0, c1, c2)
   else:
      // in-place three-prime CRT merge
      coeff ← crtCombine3(r0, r1, r2)

   // 5. Repack coefficients into 64-bit limbs with carry
   result ← packAndCarry(coeff, chunkBits = 16)

   // 6. Trim any leading zero limbs
   while size(result) > 1 and last(result) == 0:
```

```
    pop_back(result)

  return result
```

Below is the source for the NTT 32-bit version.

```
namespace ntt32 {
      /*********************************************************************
        Portable three-prime NTT (radix-2) big-integer multiplier
        - same call signature as multiplyRadix4(A,B,chunkBits)

        *  multiplyNTT  -  three-prime radix-2 NTT big-integer multiplier
        *  little-endian 64-bit limbs in / out
        *  internal chunks are base 2^chunkBits (always 16 bits here)
        *  uses 3 primes < 2^31 so all NTT buffers fit in uint32_t
        *  For higher accuracy, the routine can use six primes (about 150 M digits)
      *********************************************************************/

      /* ---------- the six “friendly” primes -------------------------- */
      static constexpr uint32_t MOD32[] = {
            167772161u,   /*  5 · 2^25 + 1 */
            469762049u,   /*  7 · 2^26 + 1 */
            1224736769u,   /* 73 · 2^24 + 1 */
            2113929217u,
            2483027969u,
            2885681153u,
      };
      /* corresponding primitive roots */
      static constexpr uint32_t PR32[6] = { 3, 3, 3, 5, 3, 3 };

      /* ---------- fast 32-bit modular helpers (mod < 2^31) ----------- */
      static inline uint32_t add_mod(uint32_t a, uint32_t b, uint32_t m) {
            uint64_t s = uint64_t(a) + uint64_t(b);
            if (s >= m) s -= m;
            return uint32_t(s);
      }
      static inline uint32_t sub_mod(uint32_t a, uint32_t b, uint32_t m) {
            /* still safe in 32 bits if m < 2^32 */
            return a >= b ? a - b : uint32_t(uint64_t(m) + a - b);
      }
      static inline uint32_t mul_mod(uint32_t a, uint32_t b, uint32_t m) {
            return uint32_t((uint64_t)a * b % m);
      }
      static inline uint32_t pow_mod(uint32_t base, uint32_t exp, uint32_t mod)
      {
            uint64_t r = 1, b = base;
            while (exp) {
                  if (exp & 1) r = r * b % mod;
                  b = b * b % mod;
                  exp >>= 1;
            }
            return uint32_t(r);
      }
      static inline uint32_t inv_mod(uint32_t a, uint32_t m) { return pow_mod(a, m - 2, m);
}


      /* ---------- 64×64 → 128 helper -------------------------------- */
      static inline uint128 mul64to128(uint64_t a, uint64_t b) {
            uint64_t a_hi = a >> 32;
            uint64_t b_hi = b >> 32;
            if (a_hi == 0 && b_hi == 0)
                  return uint128(a * b);                /* no overflow, both < 2^32 */

            uint64_t a_lo = (uint32_t)a;
            uint64_t b_lo = (uint32_t)b;
```

```
        uint64_t p0 = a_lo * b_lo;
        uint64_t p1 = a_lo * b_hi;
        uint64_t p2 = a_hi * b_lo;
        uint64_t p3 = a_hi * b_hi;

        uint64_t mid = (p0 >> 32) + (p1 & 0xFFFFFFFFu) + (p2 & 0xFFFFFFFFu);
        return uint128(p3 + (p1 >> 32) + (p2 >> 32) + (mid >> 32),
              (mid << 32) | (p0 & 0xFFFFFFFFu));
  }

  inline uint64_t mul_mod64(uint64_t a, uint64_t b, uint64_t m) {
        return mul64to128(a, b) % m;
  }
  inline uint64_t pow_mod64(uint64_t base, uint64_t exp, uint64_t mod)
  {
        uint128 r = 1, b = base;
        while (exp) {
              if (exp & 1) r = r * b % mod;
              b = b * b % mod;
              exp >>= 1;
        }
        return (uint64_t)r;
  }
  inline uint64_t inv_mod64(uint64_t a, uint64_t m) { return pow_mod64(a, m - 2, m); }

  /* ---------- radix-2 iterative NTT on vector<uint32_t> ---------- */
  static void ntt(std::vector<uint32_t>& a, int k, bool invert)
  {
        const uint32_t mod = MOD32[k];
        const uint32_t root = PR32[k];
        const int n = int(a.size());

        /* bit-reverse permutation */
        for (int i = 1, j = 0; i < n; ++i) {
              int bit = n >> 1;
              for (; j & bit; bit >>= 1) j ^= bit;
              j ^= bit;
              if (i < j) std::swap(a[i], a[j]);
        }

        /* Cooley-Tukey loops */
        for (int len = 2; len <= n; len <<= 1) {
              uint32_t wlen = pow_mod(root, (mod - 1) / len, mod);
              if (invert) wlen = inv_mod(wlen, mod);
              for (int i = 0; i < n; i += len) {
                    uint32_t w = 1;
                    for (int j = 0; j < len / 2; ++j) {
                          uint32_t u = a[i + j];
                          uint32_t v = mul_mod(a[i + j + len / 2], w, mod);
                          a[i + j] = add_mod(u, v, mod);
                          a[i + j + len / 2] = sub_mod(u, v, mod);
                          w = mul_mod(w, wlen, mod);
                    }
              }
        }

        if (invert) {
              uint32_t invn = inv_mod(n, mod);
              for (uint32_t& x : a) x = mul_mod(x, invn, mod);
        }
  }

  // ---------- convolution under one modulus
  std::vector<uint32_t> mul_mod_ntt(const std::vector<uint32_t>& A,
              const std::vector<uint32_t>& B,
              int k)
  {
        const uint32_t mod = MOD32[k];
        int n = 1;
```

```
        while (n < int(A.size() + B.size())) n <<= 1;

        std::vector<uint32_t> fa(A), fb(B);   /* copies: ntt() is in-place */
        fa.resize(n);
        fb.resize(n);

        ntt(fa, k, false);
        ntt(fb, k, false);
        for (int i = 0; i < n; ++i)
              fa[i] = mul_mod(fa[i], fb[i], mod);
        ntt(fa, k, true);
        return fa;                            /* length n, values < mod */
  }

  // ---------- 64-bit safe add: dst += a * b
  static inline void add_mul64(uint64_t& dst, uint64_t a, uint32_t b)
  {
        const uint64_t a_lo = a & 0xFFFFFFFFULL;
        const uint64_t a_hi = a >> 32;

        uint64_t lo = a_lo * b;          /* < 2^62 */
        uint64_t hi = a_hi * b;          /* < 2^58 */
        dst += lo;
        if (dst < lo) ++hi;              /* carry */
        dst += hi << 32;                 /* upper bits are zero, safe */
  }

  /*******************************************************************
   *  crt_combine2  -  merge TWO (< 2^31) residue polynomials
   *
   *  r0, r1 : vectors of equal length n
   *  start  : 0 uses (MOD32[0], MOD32[1])
   *           2 uses (MOD32[2], MOD32[3]) etc.
   *
   *  Returns : vector<uint64_t> coeff such that
   *              coeff[i] == r0[i] (mod p0)
   *              coeff[i] == r1[i] (mod p1)
   *            and 0 <= coeff[i] < p0*p1  (< 2^62)
   *
   *  Formula  : t0 = r0[i]
   *             t1 = ((r1[i] - t0) * p0^{-1}) mod p1
   *             coeff = t0 + p0 * t1
   *******************************************************************/
  static std::vector<uint64_t> crt_combine2(const std::vector<uint32_t>& r0,
              const std::vector<uint32_t>& r1,
              int start = 0)
  {
        const int      n = int(r0.size());
        const uint64_t p0 = MOD32[start + 0];
        const uint64_t p1 = MOD32[start + 1];

        const uint32_t inv_p0_mod_p1 =
              inv_mod(uint32_t(p0 % p1), uint32_t(p1));

        std::vector<uint64_t> out(n);
        for (int i = 0; i < n; ++i) {
              uint64_t t0 = r0[i];                           /* 0 .. p0-1 */
              uint64_t diff = (r1[i] + p1 - t0 % p1) % p1;     /* (r1 - t0) mod p1 */
              uint64_t t1 = mul_mod(uint32_t(diff),
                    inv_p0_mod_p1,
                    uint32_t(p1));

              uint64_t x = t0;                                 /* x = t0 + p0*t1 */
              add_mul64(x, p0, uint32_t(t1));                  /* safe 64-bit add */
              out[i] = x;                                      /* < p0*p1 < 2^62 */
        }
        return out;
  }
```

```
/*****************************************************************
 *  crt_combine64  -  merge THREE residue vectors whose moduli are
 *                      M0 = p0*p1   (< 2^63)
 *                      M1 = p2*p3
 *                      M2 = p4*p5
 *
 *  Each rX[i] is already < MX.
 *  Returns coeff[i] < M0*M1  (about 57 bits with 16-bit chunks)
 *****************************************************************/
static std::vector<uint64_t>
      crt_combine64(const std::vector<uint64_t>& r0,
            const std::vector<uint64_t>& r1,
            const std::vector<uint64_t>& r2,
            uint64_t M0, uint64_t M1, uint64_t M2)
{
      const size_t   n = r0.size();
      const uint64_t M0M1 = (uint128)M0 * M1 % M2;   /* (M0*M1) mod M2 */

      const uint64_t invM0_mod_M1 = inv_mod64(M0 % M1, M1);
      const uint64_t invM0M1_mod_M2 = inv_mod64(M0M1, M2);

      std::vector<uint64_t> out(n);
      for (size_t i = 0; i < n; ++i) {
            uint64_t t1 = r0[i];                                    /* 0 .. M0-1 */
            uint64_t t2 = mul_mod64((r1[i] + M1 - t1 % M1) % M1,
                  invM0_mod_M1, M1);

            uint128  x = uint128(t1) + uint128(M0) * t2;        /* < 2^120 */
            uint64_t x_mod = (uint64_t)(x % M2);

            uint64_t t3 = mul_mod64((r2[i] + M2 - x_mod) % M2,
                  invM0M1_mod_M2, M2);

            x += uint128(M0) * M1 * t3;                         /* exact */
            out[i] = (uint64_t)x;                               /* about 57 bits */
      }
      return out;
}

/* ===============================================================
   multiplyNTT32  (3 or 6 primes)
   Handles numbers up to roughly 40 M decimal digits with 3 primes
   or ~150 M digits with 6 primes
   =============================================================== */
std::vector<iptype>
      multiplyNTT32(const std::vector<iptype>& A,
            const std::vector<iptype>& B,
            bool sixprimes = false)
{
      const int      chunkBits = 16;          /* fixed to 16 here */
      const int      chunkPerLimb = sizeof(iptype) * 8 / chunkBits;
      const uint64_t mask = (uint64_t(1) << chunkBits) - 1;

      /* split limbs -> little-endian chunks (vector<uint32_t>) */
      auto toChunks = [&](const std::vector<iptype>& in) {
            std::vector<uint32_t> out;
            out.reserve(in.size() * chunkPerLimb);
            for (uint64_t w : in)
                  for (int i = 0; i < chunkPerLimb; ++i)
                        out.push_back(uint32_t((w >> (i * chunkBits)) & mask));
            while (out.size() > 1 && out.back() == 0) out.pop_back();
            return out;
            };
      std::vector<uint32_t> aC = toChunks(A);
      std::vector<uint32_t> bC = toChunks(B);

      /* first three convolutions */
      std::vector<uint32_t> r0 = mul_mod_ntt(aC, bC, 0);
      std::vector<uint32_t> r1 = mul_mod_ntt(aC, bC, 1);
```

```
        std::vector<uint32_t> r2 = mul_mod_ntt(aC, bC, 2);

        std::vector<uint64_t> coeff(r0.size());

        if (sixprimes) {
              /* extra three convolutions */
              std::vector<uint32_t> r3 = mul_mod_ntt(aC, bC, 3);
              std::vector<uint32_t> r4 = mul_mod_ntt(aC, bC, 4);
              std::vector<uint32_t> r5 = mul_mod_ntt(aC, bC, 5);

              /* level 1: pairwise CRT */
              std::vector<uint64_t> A2 = crt_combine2(r0, r1, 0);   /* p0*p1 < 2^62 */
              std::vector<uint64_t> B2 = crt_combine2(r2, r3, 2);   /* p2*p3 < 2^62 */
              std::vector<uint64_t> C2 = crt_combine2(r4, r5, 4);   /* p4*p5 < 2^62 */

              /* level 2: three-way merge */
              uint64_t M0 = uint64_t(MOD32[0]) * MOD32[1];          /* < 2^62 */
              uint64_t M1 = uint64_t(MOD32[2]) * MOD32[3];          /* < 2^63 */
              uint64_t M2 = uint64_t(MOD32[4]) * MOD32[5];          /* < 2^63 */

              coeff = crt_combine64(A2, B2, C2, M0, M1, M2);
        }
        else {
              /* in-place three-prime CRT merge (r0 -> coeff) */
              const uint64_t M0 = MOD32[0];
              const uint64_t M1 = MOD32[1];
              const uint64_t M2 = MOD32[2];
              const uint64_t M0M1 = M0 * M1;

              const uint32_t invM0_modM1 = inv_mod(uint32_t(M0 % M1), uint32_t(M1));
              const uint32_t invM0M1_modM2 = inv_mod(uint32_t(M0M1 % M2), uint32_t(M2));

              for (size_t i = 0; i < r0.size(); ++i) {
                    uint32_t t1 = r0[i];
                    uint32_t t2 = mul_mod((r1[i] + uint32_t(M1) - t1 % uint32_t(M1)),
                          invM0_modM1, uint32_t(M1));

                    uint64_t tmp = t1 + M0 * t2;

                    uint32_t t3 = mul_mod((r2[i] + uint32_t(M2) - tmp % uint32_t(M2)),
                          invM0M1_modM2, uint32_t(M2));

                    uint64_t x = t1;
                    add_mul64(x, M0, t2);
                    add_mul64(x, M0M1, t3);
                    coeff[i] = x;
              }
        }

        /* carry pass -> 64-bit limbs ---------------------------------- */
        size_t need = aC.size() + bC.size();
        std::vector<iptype> limbs;
        limbs.reserve((need + chunkPerLimb - 1) / chunkPerLimb);

        uint64_t carry = 0, limb = 0;
        int      bits = 0;
        for (size_t i = 0; i < need; ++i) {
              uint64_t val = coeff[i] + carry;
              uint64_t chunk = val & mask;
              carry = val >> chunkBits;
              limb |= chunk << bits;
              bits += chunkBits;
              if (bits >= 64) {
                    limbs.push_back(limb);
                    limb = 0;
                    bits = 0;
              }
        }
        if (bits || carry) {
```

```
                limb |= carry << bits;
                limbs.push_back(limb);
            }
            while (limbs.size() > 1 && limbs.back() == 0) limbs.pop_back();
            return limbs;                                   /* LSD limb first */
        }

}   /* namespace ntt32 */
```

NTT Code Comments

The code above allows you to choose either three or six primes. Regardless, the functionality is the same. Looking at just the three-primes code, we can better see the working of the NTT method, and the similarities to the FFT-based method are striking. The NTT multiplication is done in 6 steps:

Step 1. Pick your prime and padding length

For each prime p_k (you have three of them), we look at the two input chunk vectors A and B. Let n be the smallest power of two $\geq$ len(A)+len(B). Zero-pad both A and B up to length n.

Step 2. Forward NTT (the "frequency domain")

Reorder the n entries in bit-reversed order (just like FFT's bit-reversal).
Do the in-place Cooley–Tukey loops, except every addition/subtraction and "butterfly" multiply is done modulo p_k.
At the end, you've turned A and B into two "NTT spectra" $NTT_k(A)$ and $NTT_k(B)$, each still of length n.

Step 3. Pointwise multiply

For each index $i \in [0..n-1]$, compute $C_k[i] = NTT_k(A)[i] \cdot NTT_k(B)[i] \mod p_k$. No carries, no loops, just one modular multiply per entry.

Step 4. Inverse NTT (back to "time domain")

Run the same Cooley–Tukey loops, but with the modular inverse of your root of unity, then multiply every entry by $n^{-1} \mod p_k$. The result is the length-n vector of raw convolution coefficients, all reduced mod p_k.

Step 5. Repeat for all three primes

Do steps 1–4 under prime p_0 to get C_0, p_1 to get C_1, and p_2 to get C_2.

Step 6. Reconstruct the true coefficients via CRT

For each position *i*, you now have three residues ($C_0[i]$, $C_1[i]$, $C_2[i]$).
Apply the two-step CRT merge (see below, first merge C_0/C_1 into u, then merge u/C_2 into the final coefficient). That gives you the convolution result in full precision (mod $p_0p_1p_2$), which you then carry-propagate back into 64-bit limbs.

The Chinese Remainder Theorem (CRT)

The Chinese Remainder Theorem (CRT) in step 6, take the three partial convolution results from:

$$r_0[i]=C_i \bmod p_0,\ r_1[i]=C_i \bmod p_1,\ r_2[i]=C_i \bmod p_2$$

And recover the actual coefficient C_i modulo $P=p_0\,p_1\,p_2$. Here's a step-by-step recipe that a three-prime NTT implementation typically follows:

Step 1. Prepare your primes and inverses for CRT

Let p_0, p_1, p_2 be three pairwise-coprime 32-bit primes. Precompute:

$$P_{01}=p_0\cdot p_1,\ \ inv0=p_0^{-1} \bmod p_1,\ \ inv01=P_{01}^{-1} \bmod p_2$$

(inv0 is the modular inverse of p0 modulo p_1, and inv01 is the inverse of p_0p_1 modulo p_2)

Step 2. First merge: combine r_0 and r_1

We build an intermediate value u_i that satisfies

$u_i \equiv r_0[i] (\bmod p_0),\ u_i \equiv r_1[i] (\bmod p_1)$.

Compute for each index i:

$$\delta=(r_1[i]-r_0[i]) \bmod p_1,\ \ k=(\delta\cdot inv0) \bmod p_1,\ \ u_i=r_0[i]+p_0\cdot k$$

By construction, u_i is in the range [0, p0p1) and agrees with both residues.

Step 3. Final merge: combine u_i with r_2

Now lift u_i (which so far is correct mod p_0p_1) to the full modulus $P=p_0p_1p_2$. For each i:

$$\delta'=(r_2[i]-(u_i \bmod p_2)) \bmod p_2,\ \ k'=(\delta'\cdot inv01) \bmod p_2,\ \ C_i=u_i+(p_0p_1)\cdot k'$$

This final C_i sits in $[0, p_0p_1p_2)$ and satisfies:

$C_i \equiv u_i (\text{mod } p_0 p_1)$, $C_i \equiv r_2[i] (\text{ mod } p_2)$, so it simultaneously matches all three original residues.

Why the CRT works

At each stage, we correct the partial result by the "difference" between what we have and what the next prime demands, scaled by the modular inverse. Because p_0, p_1, p_2 are coprime, there's exactly one solution modulo their product. This two-step merge costs only a few multiplications and additions per coefficient, far cheaper than doing another NTT.

Why does it "feel" like FFT? You have a forward transform → pointwise multiply → inverse transform and the only difference is that every operation lives in Z/p_k space instead of C(omplex). No floating-point, no rounding, everything is exact until you do the CRT

An NTT 64-bit version is similar but needs to be able to deal with division and modulo of 128 bits, which do not usually have native support in the hardware, making it considerably slower, as seen in the performance section. However, it does have the benefit that it can handle a much larger operand size of more than 600M digits when just using a single 64-bit prime, and astronomically higher when using three 64-bit primes.

Pros and Cons of NTT vs FFT Method

Aspect	FFT (floating-point)	NTT (modular arithmetic)
Precision	Inexact (rounding error)	Exact (modulo prime)
Domain	Complex numbers (double/long double)	Finite field: integers modulo a prime
Roots of unity	Any power-of-two size (with rounding)	Requires special primes and primitive roots
Transform size	Flexible, but unstable for very large sizes	Limited by modulus and root-of-unity order
Performance	Fast with SIMD/FFTW hardware instructions	Often faster with small primes and custom code
Post-processing	Needs carry propagation + rounding	Needs CRT if using more than one prime Needs carry propagation + rounding
Memory	Large for high-precision	Compact if using 32-bit primes

In short, FFT is more straightforward and more flexible, but suffers from floating-point limitations. NTT provides exact results and often achieves faster runtime at large scales, but requires more setup.

NTT Limitations with Primes less than 2^{32}
When using primes under 2^{32}, the modulus fits in a uint32_t (32-bit), which means everything, including multiplication, reduction, and root-of-unity operations, can use native 64-bit arithmetic for intermediate values without overflow.

Why do we need multiple primes to make NTT useful?
One 32-bit prime can only represent products up to ~2^{32}, which is insufficient for serious multiplication. A product of two large integers can be thousands or millions of bits. However, when performing a convolution modulo multiple 32-bit primes, we need to reconstruct the result with the CRT.

Practical choices:
1 prime → Good for modular convolution (e.g., inside NTT-based polynomial multiplication) but not useful in practice.
3 primes → Covers 96-bit dynamic range: good enough for operands up to ~100M decimal digits. However, three selected primes represent a slightly smaller number of bits, making the maximum operand around 40M digits.
From a practical point of view, six primes can increase the operand size to approximately 160M decimal digits, but at the expense of more overhead and with a larger range.
More than six primes can extend the limitation beyond the 160M threshold. However, it will become increasingly more challenging to find useful primes and require more processing time. The downside of adding more primes is that each additional prime adds a complete transform pass + CRT recombination stage.

NTT Limitations with Primes less than 2^{64}
Using primes slightly less than 2^{64} is appealing because you can represent a broader range in a single modulus. But there are drawbacks:

Pros:
- Only one prime → No CRT step needed.
- Each limb can be larger (up to 64 bits).
- Better for very large operands (100M+ digits).

Cons:
- Arithmetic becomes trickier: you need 128-bit intermediates for multiplication and modulo.
- Only a few suitable primes exist below 2^{64} that support large enough roots of unity.
- Modular reduction (e.g., Barrett or Montgomery method) becomes more complex and heavier.

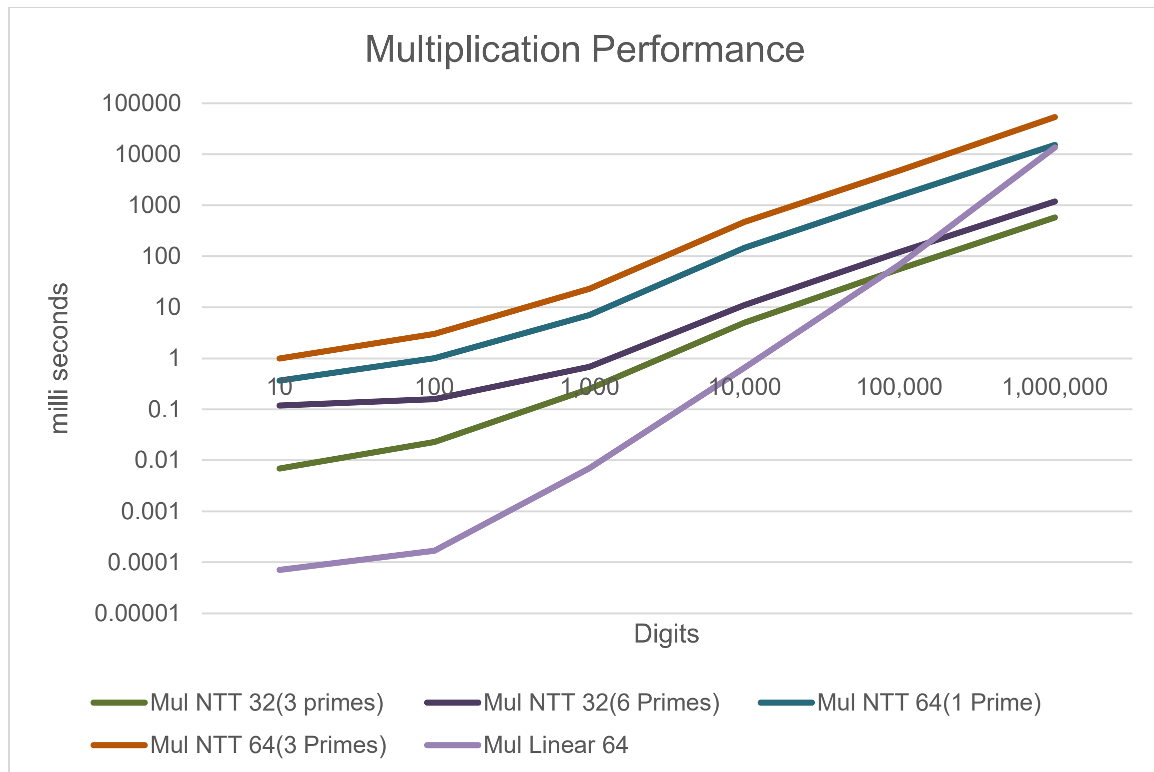
As a result, three primes under 2^{32} are often preferred for raw speed, even though a single 64-bit prime would be simpler.

Performance: Why three 32-bit primes can outperform a single 64-bit prime
Because each 32-bit modulus fits comfortably within a 64-bit register, every multiplication and addition operation remains in native hardware. You never need slow,

multi-step 128-bit routines or external big-integer libraries. Smaller numbers also mean smaller "butterfly" operations in each NTT stage; everything stays in line and tight. Modern CPUs have super-fast 64-bit integer units, so doing three rounds of 32-bit modular math beats one round of 128-bit math every time.

In real-world tests, a single NTT operation under a 64-bit prime requires more clock cycles. By contrast, running three separate NTTs under 32-bit primes is roughly three times quicker per transform, and the final Chinese Remainder recombination is just a handful of fast multiplies and adds (especially if you precompute the inverses). Even counting that extra CRT step, the total work is less.
Using three primes under 2^{32} is often 2-3 times faster than using a single prime under 2^{64}, particularly for input sizes in the range from 10^5 to 10^7 limbs.

NTT Performance of 32-bit and 64-bit



The NTT 32-bit version is significantly faster than the corresponding 64-bit version. On the 32-bit side, you incur a slight performance loss when transitioning from a 3-prime version to a 6-prime version (a drop of a factor of 2), which is expected since we are using twice as many primes. However, you can handle up to 160 million decimal digits, versus approximately 40 million decimal digits in the three-prime version. When switching to the NTT 64-bit version, we observe a significant drop in performance, despite dealing with only one 64-bit prime versus three 32-bit primes. The Mul Linear 64 is a simple linear convolution shown for comparison. You notice that NTT will perform better than just using the linear convolution approach around 100,000 digits.

Schönhage-Strassen Theory vs Practice

The Schönhage-Strassen algorithm, published in 1971 by Schönhage and Strassen, represents a theoretical step forward in computational number theory. By achieving $O(n\cdot \log(n)\cdot \log(\log n))$ complexity for integer multiplication. It remained the asymptotically fastest known method for over three decades. However, going from theoretical to practical implementation reveals computational compromises between academic theory and real-world software engineering. An interesting reference is found in [6]

The algorithm's theoretical foundation rests on the insight that integer multiplication can be transformed into polynomial multiplication through careful encoding, and polynomial multiplication can be computed efficiently. However, the issue lies in the details of implementation, where the mathematical framework collides with the realities of finite precision arithmetic, memory hierarchies, and the fundamental chicken-and-egg problems that arise when trying to bootstrap efficient arithmetic operations.

The Mathematical Foundation

The Schönhage-Strassen algorithm operates by representing large integers as polynomials and performing multiplication in the ring $Z/(2^k + 1)$. Given two n-bit integers A and B, the algorithm proceeds through several key steps:

First, the integers are decomposed into smaller "digits" in a carefully chosen base, creating polynomial representations. For an integer A with bits $a_0, a_1, \ldots, a_{n-1}$, we can write $A = \Sigma_i\, a_i \cdot 2^i$. By grouping consecutive bits, we obtain $A = \Sigma_j\, A_j \cdot \beta^j$ where $\beta = 2^w$ for some word size w, and each A_j represents a w-bit "digit."

The multiplication $A \cdot B$ then becomes a convolution of the coefficient sequences, which can be computed efficiently using the Discrete Weighted Transform (DWT) - essentially an FFT adapted for integer arithmetic. The key is that this convolution can be computed in $O(n\cdot \log n)$ time if we work in an appropriate ring.

The choice of ring $Z/(2^k + 1)$ is crucial. This ring has the property that $2^k \equiv -1 \pmod{2^k + 1}$, which creates the necessary cancellation properties for the FFT-like transform to work correctly. The transform exploits roots of unity in this ring, specifically powers of 2 that satisfy certain algebraic relationships modulo $2^k + 1$.

Complexity Analysis

The theoretical complexity breakdown of the algorithms:
- Polynomial representation: $O(n)$
- Forward transform: $O(n\cdot \log n)$
- Pointwise multiplication: $O(n)$
- Inverse transform: $O(n\cdot \log n)$
- Coefficient reduction: $O(n\cdot \log(\log n))$

The log(log n) factor emerges from the recursive nature of the arithmetic operations within the ring $Z/(2^k + 1)$. Since arithmetic in this ring requires operations on k-bit numbers, and k grows proportionally with n, the algorithm recursively applies itself to perform these ring operations, leading to the characteristic log(log n) term.

The Recursive Structure

What makes Schönhage-Strassen particularly smooth is its recursive self-similarity. The algorithm needs to perform arithmetic in $Z/(2^k + 1)$, which requires multiplication of k-bit numbers. For sufficiently large k, this multiplication is itself performed using Schönhage-Strassen recursively. This creates a mathematical structure where the algorithm bootstraps itself across multiple levels of abstraction.

The Implementation Challenge

The most fundamental challenge in implementing Schönhage-Strassen be described as a computational chicken-and-egg problem. To multiply two n-bit numbers efficiently, the algorithm requires arithmetic operations in the ring $Z/(2^k + 1)$, where k must be at least $2n + \log_2(n)$ bits to prevent coefficient overflow.

Consider the concrete requirements:
- For 128-bit multiplication: $k \geq 263$ bits
- For 1,024-bit multiplication: $k \geq 2{,}058$ bits
- For 4,096-bit multiplication: $k \geq 8{,}204$ bits

Performing arithmetic in $Z/(2^k + 1)$ requires efficient division and modular reduction by $2^k + 1$. But implementing division for k-bit numbers is precisely the problem we're trying to solve with fast multiplication. This creates a circular dependency: we need fast multiplication to implement the ring operations, but we need the ring operations to implement fast multiplication.

Ring Size Explosion

The ring size requirements grow dramatically with input size, creating a cascade of implementation challenges. For the algorithm to be practical, we need efficient methods for:

1. Modular reduction by $2^k + 1$ for very large k
2. Division by $2^k + 1$, which requires implementing division for numbers potentially twice the size of our target multiplication
3. Storage and manipulation of intermediate results that may be several times larger than the original operands

The storage requirements alone can be prohibitive. A 10,000-bit multiplication might require ring operations in $Z/(2^{20,000} + 1)$, meaning intermediate calculations involve

20,000-bit numbers. Modern computers struggle with such sizes not just due to computation time, but also due to memory bandwidth and cache behavior.

Precision and Numerical Stability

Even if we could implement exact arithmetic in the required rings, numerical stability becomes a critical concern. The FFT-like transforms amplify small errors, and the recursive structure of the algorithm compounds these effects. Traditional floating-point FFT implementations suffer from accumulated rounding errors that can corrupt the final result for large problem sizes.

The bit-reversal permutations and complex arithmetic required by the transforms introduce additional sources of error. Each butterfly operation in the FFT involves multiple floating-point operations, and the cumulative effect across thousands or millions of such operations can be devastating to accuracy.

Practical Compromises and Hybrid Approaches

The Reality of Schönhage-Strassen Implementations

Most software libraries claiming to implement Schönhage-Strassen employ sophisticated compromises that bear little resemblance to the theoretical algorithm. These implementations typically follow a hierarchical approach:

1. Classical multiplication ($O(n^2)$) for small numbers (< 100 bits)
2. Karatsuba algorithm ($O(n^{1.585})$) for medium numbers (100-1,000 bits)
3. Toom-Cook variants ($O(n^{1.465})$) for larger numbers (1,000-10,000 bits)
4. FFT/NTT-based methods for very large numbers (> 10,000 bits)

The FFT/NTT-based methods in step 4 are usually one of several alternatives, none of which is the true Schönhage-Strassen.

Many libraries use traditional complex-number FFT with careful precision management. Large integers are split into smaller pieces (typically 16-bit or 24-bit chunks) to maintain accuracy within double-precision floating-point arithmetic. This approach achieves $O(n \cdot \log n)$ complexity but sacrifices the theoretical guarantees of exact arithmetic.

Rather than working in $Z/(2^k + 1)$, some of the implementations use the Number Theoretic Transform (NTT) over finite fields Z/pZ where p is a carefully chosen prime. Common choices include primes of the form $p = k \cdot 2^n + 1$, such as $p = 2013265921 = 15 \cdot 2^{27} +$ This eliminates the ring size explosion problem but requires multiple NTT computations with different primes, followed by Chinese Remainder Theorem reconstruction.

Some implementations utilize variants such as the Discrete Hartley Transform or mixed-radix FFT approaches, which reduce computational overhead while maintaining reasonable complexity bounds.

The Bootstrap Problem in Practice

The few attempts at implementing "true" Schönhage-Strassen must solve the bootstrap problem through recursive construction. The typical approach involves:

1. Starting with classical or Karatsuba multiplication for small rings
2. Using these to implement slightly larger ring operations
3. Recursively building up to the required ring size

However, this creates enormous constant factors. Each level of recursion adds significant overhead, and the total constant factor can be so large that the asymptotic advantage disappears for all practical problem sizes. Some academic implementations report constant factors exceeding 10,000, making the algorithm slower than simpler methods even for numbers with millions of bits.

Cache Behavior and Memory Hierarchy

Modern computer architectures add another layer of complexity. The Schönhage-Strassen algorithm's memory access patterns are often cache-unfriendly, particularly during the bit-reversal phases of the FFT and when working with very large rings. The algorithm may touch memory in patterns that cause frequent cache misses, degrading performance below theoretical predictions.

In contrast, algorithms like Toom-Cook, while asymptotically slower, often exhibit better cache locality and can outperform Schönhage-Strassen on real hardware even for relatively large inputs.

Modern Library Implementations

Today's high-performance arbitrary precision libraries reflect the lessons learned from decades of implementation experience:

GMP (GNU Multiple Precision Library) uses a sophisticated hierarchy of algorithms with carefully tuned crossover points. Its Schönhage-Strassen implementation is a truncated FFT with floating-point arithmetic, typically engaging only for numbers exceeding 10,000-20,000 bits.

FLINT (Fast Library for Number Theory) emphasizes NTT-based approaches, using multiple prime moduli and Chinese Remainder Theorem reconstruction. This avoids many of the pitfalls of working in $Z/(2^k + 1)$ while maintaining good asymptotic performance.

MPIR and other libraries often employ hybrid approaches, combining multiple techniques and selecting the optimal method based on input size, available precision, and hardware characteristics.

The Theory-Practice Divide

The Schönhage-Strassen algorithm exemplifies both the power and limitations of theoretical computer science. However, the implementation challenges reveal fundamental tensions between theoretical models and practical computation. The algorithm's reliance on exact arithmetic in giant rings creates bootstrap problems that are difficult or impossible to solve efficiently. The resulting implementations either sacrifice exactness (through floating-point approximations) or efficiency (through recursive bootstrap constructions with massive constant factors).

For practitioners implementing high-performance arithmetic, the lesson is clear: asymptotic optimality does not guarantee practical superiority. The most effective implementations often combine multiple techniques, carefully tuned to the specific requirements of target applications and hardware platforms.

Fürer's Method

In 2007, Martin Fürer published an algorithm achieving $O(n \log n\, 2^{O(\log n)})$ complexity [8], technically surpassing the Schönhage-Strassen asymptotic bound. However, the improvement only becomes apparent for numbers with more bits than there are atoms in the observable universe, highlighting the disconnect between theoretical advances and practical relevance.

More recently, Harvey and van der Hoeven achieved $O(n \log n)$ complexity using a refined analysis of the FFT over complex numbers; however, the crossover point still lies far beyond practical computation ranges.

Choosing the right method in real projects

If you ask for a general recommendation, you will usually get something like this. However, in many cases, it differs dramatically based on the target machine and the level of code optimization.

As we will see under the performance section, using my arbitrary precision implementation, I get a different set of recommendations.

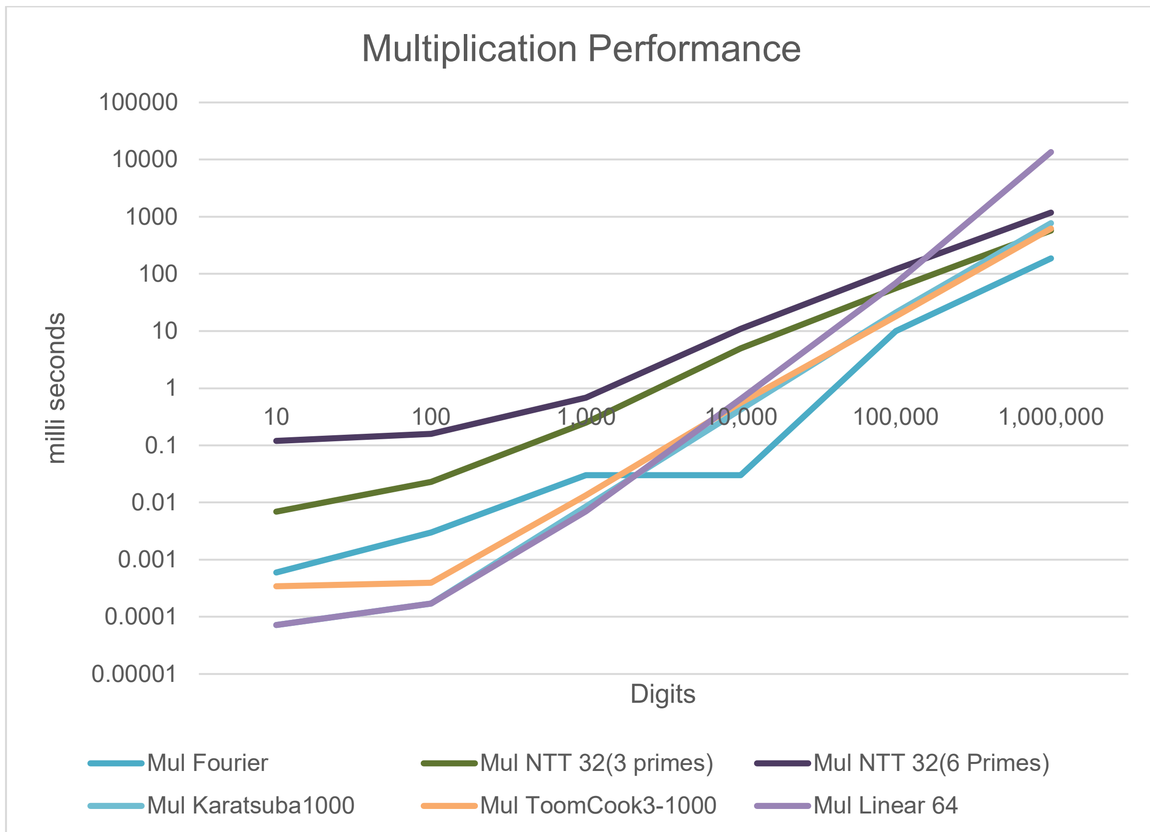
Limb range (rough guide)	Preferred algorithm	Reason

1 – 200	Schoolbook	The overhead of the others outweighs the gains
200 – 4,000	Karatsuba	Lower work, small code, good cache hit rate
4 ,000 – 50,000	Toom-3 (or Toom-4)	Best balance of work vs. overhead
50,000 – 1,000,000	Floating-point FFT	*n log n* time, simple code, wide SIMD
1,000,000 +	Schönhage–Strassen or high-quality NTT	Avoids rounding, mature code available.
1,000,000,000 +	Fürer-type	Asymptotically minimal work

These cut-offs shift with processor cache sizes, SIMD width, and the level of optimization in your code, so always benchmark on the target machine.

Performance

Below is the result of measuring performance for multiplying numbers with different operand sizes from 10 decimal digits to 1 million decimal digits.



As we can see from the performance chart, multiplying using a 64-bit linear convolution is by far the fastest method below approximately 4,000 decimal digits. (Karatsuba has similar performance as linear convolution from 1000 to 5,000) After that, the optimized Fast Fourier Transformation (special FFT radix 2) takes over and maintains its best performance, extending to over 1 million digits in this performance chart. The FFT radix 2 is an optimized version from [4] that avoids the complex arithmetic library and bases the transformation solely on regular 64-bit floating-point arithmetic. After the optimized FFT Fourier, the ToomCook3 becomes slightly faster than Karatsuba above the 10,000-digit mark.

Conclusion

This paper presents a discussion of the practical implementation of fast integer multiplication algorithms for arbitrary-precision numbers. All of these algorithms can be seen in action [9]. Because each multiplication method has different performance characteristics, you will end up implementing a hybrid approach where simpler algorithms are used for smaller numbers (linear convolution) and more advanced methods are used for medium-sized numbers, e.g., Karatsuba, Toom-Cook 3. For large numbers, the FFT and NTT methods are employed.

Recommendation for arbitrary multiplication implementation

- Store all big integers in a C++ std::vector<uint64_t> with limb 0 the least-significant 64 bits.
- Maintain a simple linear convolution routine for the smallest sizes and use it as the base case for recursive schemes.
- Add Karatsuba and Toom-3 for the middle ground.
- For very large operands, build either an FFT (fast to code, but watch rounding) or an NTT/Schönhage–Strassen (heavier code, exact result).
- The cutting-edge n·log n algorithms (Schönhage-Strassen, Fürer) are still not widely in use and hard to implement efficiently, but mark the direction of future record-size multiplication algorithms.

Reference

1. Karatsuba algorithm - Wikipedia
2. Toom–Cook multiplication - Wikipedia
3. Schönhage–Strassen algorithm - Wikipedia
4. Numerical recipes in C++, 3rd edition, Cambridge University Press, New York, NY, 2007
5. Wilkinson, J H, Rounding errors in Algebraic Processes, Prentice-Hall Inc, Englewood Cliffs, NJ, 1963

6. J. Schulz, A verified functional implementation of the Schönhage-Strassen-Algorithm, Department of Mathematics, TUM School of Computation, Information Technology, Technical University of Munich.
7. Fast Fourier transform - Wikipedia
8. M. Fürer. Fast Integer multiplication - Webarchive
9. A completely arbitrary precision library in C++. Includes arbitrary integer, floating point, fractions, and interval arithmetic. Arbitrary Precision C++ Packages